REPORT DOCUMENTATION PAGE

1. Recipient's Reference	2. Originator's Reference	3. Further Reference	4. Security Classification of Document
	AGARD-AG-289	ISBN 92-835-1554-4	UNCLASSIFIED

5. Originator	Advisory Group for Aerospace Research and Development North Atlantic Treaty Organization 7 rue Ancelle, 92200 Neuilly sur Seine, France
6. Title	FAULT TOLERANT CONSIDERATIONS AND METHODS FOR GUIDANCE AND CONTROL SYSTEMS
7. Presented at	

8. Author(s)/Editor(s)	9. Date
Editor: M. l'Ingénieur Général Marc J.Pelegrin	July 1987

10. Author's/Editor's Address	11. Pages
Directeur du Centre d'Etudes et de Recherches de Toulouse, ONERA, 2 Avenue Edouard Belin BP 4025, F-31055 Toulouse, Cedex, France	136

12. Distribution Statement	This document is distributed in accordance with AGARD policies and regulations, which are outlined on the Outside Back Covers of all AGARD publications.

13. Keywords/Descriptors

High integrity software	Fault diagnostic
Fault tolerance, fault avoidance software	Error recovery
Software dependability	Dissimilar redundancy

14. Abstract

This AGARDograph, prepared and edited at the request of the Guidance and Control Panel of AGARD, presents a description of recent trends and developments in analysis and design methods for fault tolerant guidance and control systems architectures. The problems and issues associated with these architectures, including both hardware and software aspects, are addressed exploring the many developments underway in NATO in the following areas: Advances in fault tolerant architectures; Advances in analytical fault-detection methods; Design considerations and methods; Analysis and testing methods.

This AGARDograph has been prepared at the request of the Guidance and Control Panel of AGARD.

AGARD-AG-289

AGARD-AG-289

AGARD

ADVISORY GROUP FOR AEROSPACE RESEARCH & DEVELOPMENT

7 RUE ANCELLE 92200 NEUILLY SUR SEINE FRANCE

AGARDOGRAPH No.289

Fault Tolerant Considerations and Methods for Guidance and Control Systems

NORTH ATLANTIC TREATY ORGANIZATION

DISTRIBUTION AND AVAILABILITY
ON BACK COVER

AGARD-AG-289

NORTH ATLANTIC TREATY ORGANIZATION

ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT , Paris

(ORGANISATION DU TRAITE DE L'ATLANTIQUE NORD)

AGARDograph No.289

FAULT TOLERANT CONSIDERATIONS AND METHODS FOR GUIDANCE AND CONTROL SYSTEMS

Edited by

Ingénieur Général Marc J.Pelegrin
Directeur du Centre d'Etudes et de Recherches de Toulouse
ONERA
2 Avenue Edouard Belin
BP 4025
F-31055 Toulouse Cedex
France

This AGARDograph has been prepared at the request of the Guidance and Control Panel of AGARD.

THE MISSION OF AGARD

The content of this publication has been reproduced
directly from material supplied by AGARD or the authors.

Published July 1987

Copyright © AGARD 1987
All Rights Reserved

ISBN 92-835-1554-4

*Printed by Specialised Printing Services Limited
40 Chigwell Lane, Loughton, Essex IG10 3TZ*

PREFACE

A previous AGARDograph (No 258, May 1980) dealt with a similar subject to the present one; its title was 'Guidance and Control Software'. A quick comparison of titles of papers points out that five or six years later the major concerns of today's engineers are the same. The tremendous possibilities of modern processors still hide the difficulties of providing adequate software which would be demonstrated as fault-exempt for all possible configurations of input data.

Most of the papers deal with the reliability problem through various approaches. Two papers concern applications directly but all papers give examples of application. It is worthwhile noting that hardware faults are no longer considered. However, due to the multiplicity of sensors, the detection of sensors failure would simplify the recovery process if the failure is immediately detected; if not, wrong data are sent to the processor(s) and the fault-detection process becomes much more complicated and lasts longer.

External disturbances, such as lightning effect on the behaviour of a computer (it is assumed that the physical protection of the hardware works normally) have and will have more importance. This is due to two different causes. The energy of signals used in VLSI techniques is always decreasing; indeed the size of the chips is, also, decreasing and the induction due to the discharge current (some hundreds of thousand amperes) is decreasing as well; but the connections between sensors and computers or between computers remain about the same. Thus the ratio between induction and signal energies increases.

The second cause is the rapid development of composite materials which cancel the Faraday-shielding effect. Many investigations are being carried out at the present time and the type of failure induced on the software execution in a computer is studied carefully, both on a theoretical and practical basis.

Software reliability is a permanent problem. New methods to produce modular high-reliability software will soon give rise to a new way of programming computers: they will allow the combining of these modules into a large software of about the same degree of reliability. They will also allow re-use of parts of already existing and tested softwares.

This means that a new AGARDograph on the same subject is very likely in some five years time.

CONTENTS

*(From LS.109: Fault Tolerance and Redundancy Management Techniques).

Computing Systems Dependability and Fault Tolerance: Basic Concepts and Terminology

J.C. Laprie

LAAS-CNRS
7, Avenue du Colonel Roche
Toulouse
31077 Cedex
France

Abstract

This paper provides a conceptual framework for expressing the constituents of dependable computing:

- the impairments to dependability: faults, errors, and failures;
- the means for dependability: fault avoidance, fault tolerance, fault removal, and fault forecasting;
- the measures of dependability: reliability and availability.

Emphasis is put on the dependability impairments and on fault tolerance. Informal but precise definitions are given.

Introduction

This paper is aimed at giving informal but precise definitions caracterizing the various attributes of computing systems dependability. It is a contribution to the work undertaken within the "Reliable and Fault Tolerant Computing" scientific and technical community [Jes 77, Avi 78, Ran 78, Car 79, Lap 82, And 81, Sie 82, Cri 85] in order to propose clear and widely acceptable definitions for some basic concepts.

The paper results from an elaboration on [Lap 85, Avi 86]. It proceeds by refinements: dependability is introduced as a global concept in a first section. Fault tolerance is then detailed in a second section. A glossary is given in annex which recapitulates the terms defined throughout the paper.

The guidelines which have governed this presentation can be summed up as follows:

- search for the minimum number of concepts enabling the dependability attributes to be expressed;
- use of terms which are identical to —whenever possible— or as close as possible to those generally used; as a rule, a term which has not been defined retains its ordinary sense (as given by any dictionary);
- emphasis on integration (as opposed to specialization [Gol 82]).

In each section, concise definitions are given first, then they are heavily commented in order to (attempt to) show their wide applicability. Boldface characters are used when a term is defined, italic characters being an invitation to focus the reader's attention.

The Dependability Concept

BASIC DEFINITIONS AND ASSOCIATED TERMINOLOGY

Dependability is that property of computing systems which allows *reliance to be justifiably placed on the service it delivers.*
The **service** delivered by a system is its behavior *as it is perceived* by its user(s); a **user** is another system (human or physical) which interacts with the former.

A system **failure** occurs when the delivered service deviates from the specified service, where the service **specification** is an *agreed* description of the expected service. The failure occurred because the system was erroneous: an **error** is that part of the system state —with respect to the computation process— which is liable to lead to failure. The *adjudged or hypothesized* cause of an error is a **fault**.
An error is thus the manifestation of a fault *in the system,* and a failure is the effect of an error *on the service.*

Achieving a dependable computing system calls for the *combined* utilization of a set of methods which can be classed into:

- **fault avoidance**: how to prevent, *by construction,* fault introduction;
- **fault tolerance**: how to provide, *by redundancy,* service complying with the specification in spite of faults;
- **fault removal**: how to minimize, *by verification,* the presence of faults;
- **fault forecasting**: how to estimate, *by evaluation,* the presence, the creation, and the consequences of faults.

Fault avoidance and fault tolerance may be seen as constituting dependability **procurement:** how to *provide* the system with the ability to deliver the specified service; fault removal and fault forecasting may be seen as constituting dependability **validation**: how to *reach confidence* in the system's ability to deliver the specified service.

The life of a system is perceived by its users as an alternation between two states of the delivered service with respect to the specified service:

- **proper service**, where the service is delivered *as* specified;
- **improper service**, where the delivered service *differs from* the specified service.

The events which constitute the transitions between these two states are the failure and the **restoration** of service. Quantifying the alternation between proper and improper service leads to the two main measures of dependability:

- **reliability**: a measure of the *continuous* delivery of proper service —or, equivalently, of the *time to* failure— from a reference initial instant;
- **availability**: a measure of the delivery of proper service *with respect to the alternation* of proper and improper service.

COMMENTS

1- On the Introduction of Dependability as a Global Concept

Why should another term be added to an already long list ? Reliability, availability, safety, etc. The reasons are basically two-fold:

- to remedy the existing confusion between reliability in its general meaning (reliable system) and reliability as a mathematical quantity (system reliability)[1];
- to show that reliability, maintainability, availability, safety, etc. are quantitative measures corresponding to distinct perceptions of the same attribute of a system: its *dependability.*

In regard to the term "dependability", it is noteworthy that from an etymological point of view, the term "reliability" would be more appropriate: ability to rely upon. Although dependability is synonymous to reliability, it brings in the notion of dependence at a second level. This may be felt

[1] Most books having the word "reliability" in their title actually deal with how to evaluate, measure, predict the reliability of systems, not really with how to build reliable systems.

as a negative connotation at first sight, when compared to the positive notion of trust as expressed by reliability, but it does highlight our society's ever increasing dependence upon sophisticated systems in general and especially upon computing systems. Moreover, "to rely" comes from the French "relier", itself from the Latin "religare", to bind back: re-, back and ligare, to fasten, to tie. The French word for reliability, "fiabilité", traces back to the 12th century, to the word "fiableté" whose meaning was "character of being trustworthy"; the Latin origin is "fidare", a popular verb meaning "to trust". Simultaneous consideration of the English and French origins for reliability thus leads to the (unsurprising) association of two key concepts: ties for trust.

Finally, it is interesting to note that viewing dependability as a more general concept that reliability, availability, etc. and embodying the latter terms, has already been attempted in the past (see e.g. [Hos 60]), although with less generality than here, since the goal was to define a measure.

2- On the Notions of Service and its Specification, and of System

From its very definition, the service delivered by a system is clearly an *abstraction* of its behavior. It is noteworthy that this abstraction is highly dependent on the application that the computer system supports. An example of this dependence is the important rôle played in this abstraction by the time: the time granularities of the system and of its user(s) are generally different.

Concerning specification, what is essential within the present context is that it is a description of the service which is *agreed upon* by two persons or corporate bodies —in fact, legal personnae: the system supplier (in a broad sense of the term: designer, builder, vendor, etc.) and its human user(s)[2]. The agreement is necessary in order that the specification can (at least) serve as a basis for adjudicating unambiguously whether the delivered service is acceptable or not. What precedes does not mean that a specification will not change once established. This would be simple ignorance of the facts of life, which imply *change*. The changes may be motivated by modifying the service expectation: modification of functionality, or correction of some undesired features such as deficiencies in the agreed specification. Once more, what is important is that the specification is (again) agreed upon. It must be stressed that such matters as environment conditions, exposure time duration, observability, readiness, etc., can —and should— be captured by an appropriately stated specification.

Up to now, a **system** has been considered as a whole, emphasizing its externally perceived behavior; a definition complying with this "black box" view is: an entity having interacted, interacting, or likely to interact with other entities, i.e., with other systems[3]. The **behavior** is then simply what the system *does* [Zie 76]. What *enables it to do what it does* is the **structure** of the system or its organization. Adopting the spirit of [And 81], a system, from a structural ("glassbox") viewpoint, is a set of components bound together in order to interact; a **component** is another system, etc. The recursion stops when a system is considered as being **atomic**: any further internal structure cannot be discerned, or is not of interest and can be ignored. The term "component" has to be understood in a broad sense: layers[4] of a system as well as intralayer components; in addition, a component being itself a system, it embodies the interrelation(s) of the components of which it is composed.

Based on the preceding view of system structure, the notions of service and of its specification apply equally naturally to the components[5].

[2] The agreement may be implicit, e.g. when using off-the-shelf systems.

[3] Giving recursive definitions is not for recursion's sake. The aim is to emphasize relativity with respect to the adoted viewpoint. So is it for the notion of system: a given system's boundaries may vary depending whether it is viewed by its designer(s), by its user(s), by its maintenance crew, etc.

[4] In the sense of protocols, i.e. a given layer using the services provided by lower layer(s), including hardware, and delivering services to the upper layer(s).

[5] This is especially interesting in the design process, when off-the-shelf components, either hardware or software —"reusable" software— are used: what is of actual interest to the designer is the service they are able to provide, not their detailed (internal) behavior.

3- On the Notions of Fault, Error, and Failure

The sources of faults are extremely diverse. The three main viewpoints according to which they can be classified are:

1) the phenomenological causes, which lead to distinguish [Avi 78]:
   - **physical faults**, which are due to adverse physical phenomena,
   - **human-made faults**, which result from human imperfections;
2) the system boundaries, which lead to distinguish **internal faults** and **external faults**, the latter resulting from the system's interference with its physical or human environment.
3) the temporal persistence, which leads to distinguish:
   - **permanent faults**, which are irreversible structural states,
   - **temporary faults**, who do not act longer than a certain maximum time.

The first two viewpoints lead to four classes of faults:

- physical internal faults, due to physico-chemical disorders (threshold changes, short circuits, open circuits, etc.);
- physical external faults, due to environmental perturbations (electromagnetic perturbations, radiations, temperature, vibrations, etc.);
- human-made internal faults, or **design faults**, resulting from imperfections committed either a) during the initial design of the system (broadly speaking, from requirement specification to implementation) or during subsequent modifications, or b) during the establishment of the procedures for operating or maintaining the system;
- human-made external faults, resulting from the violation of operating or maintenance procedures.

It could be argued that introducing the phenomenological causes in the classification criteria of faults may lead recursively "a long way back", e.g. why are programmers doing mistakes? why are integrated circuits failing? The very notion of fault is *arbitrary*, and is in fact a facility provided for stopping the recursion. Hence the definition given: *adjuged or hypothesized* cause of an error. This cause may vary depending upon the adopted viewpoint: fault tolerance mechanisms, maintenance engineers, repair shop, designer, semiconductor physicist, etc. In our view, recursion stops at *the cause which is intended to be avoided or tolerated.* This view provides consistency to the distinction between human-made and physical faults: a computing system is a human artifact and as such any fault in it or affecting it is ultimately human-made since it represents human inability to master all the phenomena which govern the behavior of a system. In an absolute sense, a distinction between physical faults and human-made faults (especially design faults) may be considered unnecessary; however, this distinction is of importance when considering the (current) methods and techniques for procuring and validating dependability. If the recursion mentioned above is not stopped, then *a fault is nothing other than the consequence of a failure of a system that has delivered* (including the designers) *or is now delivering a service to the considered system.*

Examples of the preceding discussion follow:

- a design fault is consecutive to a designer failure;
- a physical internal fault is due to a hardware component failure, which is itself the consequence of (an) error(s) at the electrical or electronical level, in turn originating from the hardware production, or from —the limits of— our knowledge of the semiconductor physics: the "physics reliability" community rarely characterizes failures as "sudden and nonpredictable";
- a physical or human-made external fault is in fact a design fault: the inability to foresee all the situations the system will be faced with during its operational life, or the refusal of considering some of them (e.g. for economic reasons); for instance in the case of a transient fault under the form of an electromagnetic perturbation: is it an external fault or a design fault, i.e. the lack of adequate shielding?

The temporal persistence viewpoint deserves the following comments:

1) Internal faults can be permanent or temporary, whereas external faults are generally temporary.
2) Permanent internal faults are generally due to failures, either of a hardware component or of a designer.
3) Temporary physical external faults are often termed as **transient faults**.

4) Temporary internal faults are often termed as **intermittent faults**; those faults result from the presence of rarely occurring combinations of conditions; examples are a) "pattern sensitive" faults in semiconductor memories, change in parameters of a hardware component (effect of temperature variation, delay in timing due to parasitic capacitance, etc.), or b) situations —affecting either hardware or software— occurring when system load comes above a certain level, such as marginal timing and synchronisation. In fact, the term "fault" is in such cases is actually an abstraction for *fault conditions*.

The table of Figure 1 summarizes the classes of faults with respect to the three viewpoints considered.

Phenomenological Cause		System Boundary		Temporal Persistence	
Physical	Human-made	Internal	External	Permanent	Temporary
★		★		★	
★		★			★
★			★		★
	★	★		★	
	★	★			★
	★		★		★

Figure 1 - Fault classes

Concerning human-made faults, an additional viewpoint for their classification is to consider whether they are *accidental* or *intentional*. Up to now, only accidental faults have been —implicitly— dealt with. Intentional faults, either design faults ("Trojan horses") or interaction faults (intrusions) lead directly to *security*. However, the corresponding methods and techniques will not be addressed in this paper.

An error was defined as being *liable* to lead to failure. Whether or not an error will actually lead to a failure depends on two major factors:

1) The system composition, and especially the nature of the existing redundancy:
   - *intentional* redundancy (introduced to provide fault tolerance) which is explicitly intended to prevent an error from leading to failure,
   - *unintentional* redundancy (it is practically difficult if not impossible to build a system without any form of redundancy[6]) which may have the same —unexpected— result as intentional redundancy.
2) The definition of a failure from the user's viewpoint: what is a failure for a given user may be a bearable nuisance for another one. Examples are a) accounting for the user's time granularity: an error which "passes through" the system-user(s) interface may or may not be viewed as a failure depending on the user's time granularity, b) the notion of "acceptable error rate" —implicitly before considering that a failure has occurred— in data transmission.

Based on the given definition of a system from the structural viewpoint, the discussion of whether "failure" applies to a system or a component is simply irrelevant, since a component is itself a system. When atomic systems are dealt with, the notion of an "elementary" failure comes naturally.

[6] A classical problem in hardware testing is the removal of such "false redundancies", whose effect may be to mask errors, and as such to make more complicate the test patterns generation.more complicate.

The structural view of a system enables fault pathology to be made more precise. The creation and manifestation mechanisms of faults, errors, and failures may be summarized as follows:

1) A fault may be **dormant** or **active**; a fault is active when it produces an error. A fault may cycle between its dormant and active states. Physical faults can directly affect the physical components only, whereas human-made faults may affect any component.
2) An error may be **latent** or **detected**, either by a detection algorithm or mechanism within the component. An error may, and in general does, propagate[7] from one component to another; by propagating, an error creates other —new— error(s). An error may thus originate from:
   - activation of a dormant fault within the same component,
   - propagation of an error within the same component or from another component.
3) A component failure occurs when an error affects the service delivered by the component as a response to request(s) from another component. A component failure may result in a fault for the component(s) to whom it delivers service.

These mechanisms enables the "fundamental chain" to be completed:

$$\cdots \rightarrow \text{failure} \rightarrow \text{fault} \rightarrow \text{error} \rightarrow \text{failure} \rightarrow \cdots$$

A given fault in a given component may result from different possible sources. For instance, a fault in a physical component —e.g. stuck at ground voltage— may result from:

- a physical failure (e.g. caused by a threshold change);
- an error caused by a design fault —e.g. faulty microinstruction— propagating "top-down" through the layers and causing a short between two circuit outputs for a duration long enough to provoke a short-circuit having the same consequence as the threshold change.

In conclusion, we observe that faults, errors, and failures are all undesired circumstances. Assignment of the terms fault, error, failure simply takes into account current usage: a) fault avoidance, tolerance, and diagnosis, b) error detection and correction, c) failure rate.

It has to be noted that the labels "fault" and "error" are sometimes interchanged, as in [IEE 82]. The assigment adopted here acknowledges the long-time usage of the coding theory.

4- On Fault Avoidance, Tolerance, Removal, and Forecasting

All the "how to's" which appear in the basic definitions are in fact goals which cannot be fully reached, as all the corresponding activities are human activities, and thus imperfect. These imperfections bring in *dependencies* which explain why it is only the *combined* utilization of the above methods —preferably at each step of the design and implementation process— which can lead to a dependable computing system. These dependencies can be sketched as follows: in spite of fault avoidance by means of design methodologies and construction rules (imperfect in order to be workable), faults occur; hence the need for fault removal; fault removal is itself imperfect, as are the off-the-shelf components of the system, hence the need for fault forecasting; our increasing dependence on computing systems brings in fault tolerance, which in turns necessitates further construction rules, and thus fault removal, fault forecasting, etc.

It must be noted that the process is even more recursive than it appears from the above: current computer systems are so complex that their design and implementation need computerized tools in order to be cost-effective (in a broad sense, including the capability of succeeding within an acceptable time scale). These tools have themselves to be dependable, and so on.

The preceding reasoning explains why in the given definitions fault removal and fault forecasting are gathered into validation. Validation is often limited to what has been termed as verification; in that case these two terms are often associated, e.g. "V and V" [Boe 79], the distinction being related to the difference between "building the system right" (related to verification) and "building the right system" (related to validation). What is proposed is simply an extension of this concept: the answer to the question "am I building the right system?" being complemented by "for

[7] The non reflexive form of "propagate" is intentionally used: an error does not propagate itself, it just propagates. Although "propagate" was retained due to its wide use, beter words would probably be "spread", or "breed".

how long will it be right?"[8]. In addition, fault removal is usually closely associated with fault avoidance, both constituting fault prevention. Besides highlighting the need for validating the procedures and mechanisms of fault tolerance, considering fault removal and fault forecasting as two constituents of the same activity —validation— is of great interest in that it enables a better understanding of the notion of **coverage**, and thus of an important problem introduced by the above recursion: *the validation of the validation,* or how to reach confidence in the methods and tools used in building confidence in the system. Coverage refers here to a measure of the representativity of the situations to which the system is submitted during its validation with respect to the actual situations it will be confronted with during its operational life.

5- On the Measures of Dependability

Only two basic measures (or "metrics") have been considered, reliability and availability. Usually, a third one, **maintainability** is also considered: a measure of the continuous delivery of improper service, or equivalently, of the time to restoration. This measure is no less important than those previously defined; it was not introduced in the basic definitions because it may, at least conceptually, be deduced from the other two. It is noteworthy that availability encapsulates both the frequency of failure and the duration of proper service at each alternation of proper-improper service.

The term "probability" has intentionally been not employed in the given definitions, so as to reinforce the significance of the defined measures. As the considered events are non-deterministic, random variables are associated with them, and the measures which are dealt with are probabilities.

A system may not, and generally does not, always fail in the same way. This immediately brings in the notion of the consequences of a failure upon the other systems with which the considered system is interacting, i.e. its **environment**. Several failure modes can generally be distinguished (in the specification), ordered according to the increasing severity of their consequences, i.e. their **criticality**. A special case of great interest is that of systems which exhibit two failure modes (or whose failure modes can be grouped into two classes) whose criticalities differ considerably:

- **benign failures**, where the consequences are of the same order of magnitude (generally in terms of cost) as those of the service delivered in the absence of failure;
- **malign** or **catastrophic failures**, where the consequences are non commensurable with those of the service delivered in the absence of failure.

Through grouping the states of proper service and improper service subsequent to benign failures into a safe state (in the sense of being free from damage, not from danger), the generalization of reliability leads to an additional measure: a measure of continuous safeness, or equivalently, of the time to catastrophic failure, i.e., **safety**[9]. A direct generalization of the availability, i.e. a measure of safeness with respect to the alternation of safeness and improper service after catastrophic failure, would not provide a significant measure. When a catastrophic failure has occurred, the consequences are generally so important that system restoration is not of prime importance for —at least— the two following reasons:

- it comes second to repairing (in the broad sense of the term, including legal aspects) the consequences of the catastrophe;
- the lengthy period prior to being allowed to operate the system again (investigation commissions) would lead to meaningless numerical values.

A "hybrid" reliability-availability-type measure has thus to be defined: a measure of proper service delivery with respect to the alternation of proper service and improper service after benign failure. This measure is of interest in that it provides indeed a quantification of the system availability *before* occurrence of a catastrophic failure, and as such enables quantification of the so-called "reliability- (or availability-) safety tradeoff".

[8] Validation stems from "validity", which encapsulates two notions:
- validity at a given moment, which relates to fault removal;
- validity for a given duration, which relates to fault forecasting.

[9] It is not the author's intention to contribute to the controversy whether safety includes reliability or vice-versa: the proposed answer is that both are attributes of dependability.

SUMMARY

What has been presented in this section actually constitutes an attempt to build a taxonomy of dependable computing. The concepts introduced may be gathered into three main classes of attributes:

- the **impairments** *to* dependability, which are undesired —but not unexpected— circumstances causing or resulting from un-dependability (whose definition is very simply derived from the definition of dependability: reliance cannot, or will not, be any longer justifiably placed on the service); the impairments are the faults, errors, and failures.
- the **means** *for* dependability, which are the methods, tools, and solutions enabling a) the system to be provided with the ability to deliver a service on which reliance can be placed (dependability procurement by fault avoidance and fault tolerance), and b) the user to reach confidence in this ability (dependability validation by fault removal and fault forecasting);
- the **measures** *of* dependability, which enable the service quality resulting from the impairments and the means opposing to them to be appraised; the two main measures are reliability and availability.

The dependability constituents can be represented under the form of a tree as in figure 2.

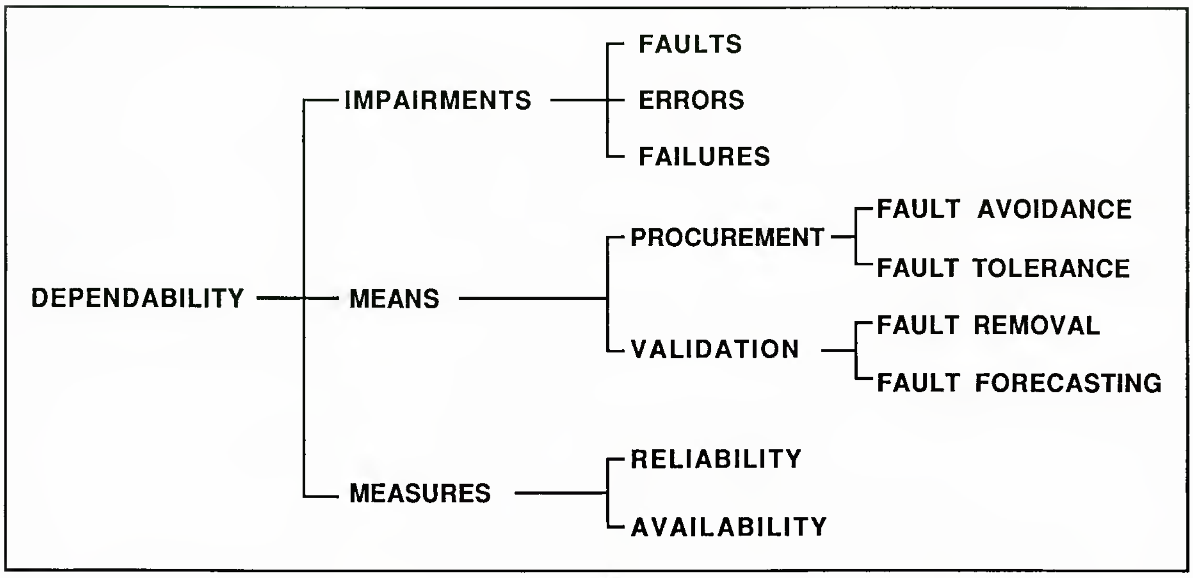


Figure 2 - The constituents of dependability

FAULT TOLERANCE

BASIC DEFINITIONS

Fault tolerance is carried out by error processing and by fault treatment [And 81]. **Error processing** is aimed at removing errors from the computational state, if possible before occurrence of failure; **fault treatment** is aimed at preventing (a) fault(s) to be activated —again.

Error processing may take on two forms:

- **error recovery**, where an error-free state is substituted for the erroneous state; this substitution may take on two two forms [And 81]:
  - **backward error recovery**, where the erroneous state transformation consists of bringing the system back to an already occupied state prior to error occurrence; it involves the establishment of **recovery points**, which are points in time during the execution of a process for which the then current state may subsequently need to be restored;

  - **forward error recovery**, where the erroneous state transformation consists of finding a new state, which was never occupied before, or which was not occupied since the same error occurred;
- **error compensation**, where the erroneous state contains enough redundancy to enable the delivery of an error-free service from the erroneous (internal) state.

When error recovery is employed, the erroneous state needs to be (urgently) identified as being erroneous prior to being transformed, which is the purpose of **error detection**. On the other hand, compensation may be applied systematically, even in the absence of error(s), then providing **fault masking**.

The first step in fault treatment is **fault diagnostics**, which consists in determining the cause(s) of error(s), in terms of both location and nature. Then come the actions aimed at fulfilling the main purpose of fault treatment: preventing the fault(s) from being activated again, thus aimed at making it(them) passive, i.e. **fault passivation**. This is carried out in removing the component(s) identified as being faulty from further execution processes. If the system is no longer capable of delivering the same service as before, then a *reconfiguration* may take place.

Fault treatment is generally complemented by **maintenance** (except of course for non-maintained systems), aimed at *removing* faults. Maintenance actions can be put into two classes:
- **corrective maintenance**, aimed at removing those faults which have produced detected errors,
- **preventive maintenance**, aimed at removing faults before they are activated.

COMMENTS

1- On the Tolerated fault classes

The preceding definitions apply to physical faults as well as to design faults: the class(es) of faults which can actually be tolerated depend(s) on the fault hypothesis which is being considered in the design process, and thus depend on the *independency* of redundancies with respect to the process of fault creation and activation. An example is provided when considering tolerance to physical faults and tolerance to design faults. A (widely used) method to attain fault tolerance is to perform multiple computations through multiple channels. When tolerance to physical faults is foreseen, the channels may be identical, based on the assumption that hardware components fail independently; such an approach is not suitable for the tolerance to design faults where the channels have to provide *identical services* through *independent designs and implementations* [Elm 72, Ran 75, Avi 78], i.e. through **design diversity** [Avi 84].

Fault tolerance is (also) a recursive concept: it is essential that the mechanisms aimed at implementing fault tolerance be protected against the faults which can affect them. Examples are voter replication, self-checking checkers [Car 68], "stable" memory for recovery programs and data [Lam 81].

2- On Error Processing and Fault Treatment

The goal of error processing is the preservation of data integrity, which in turns requires that the data contained in the components involved in this preservation to be consistent [Wen 78, Gra 78, Pea 80].

The knowledge of some system properties may limit the necessary amount of redundancy. Examples of these properties are regularities of structural nature: error detecting and correcting codes, robust data structures [Tay 80], multiprocessors and local area networks [Hay 76]. The fault that are tolerated are then dependent upon the properties which are accounted for, since they directly intervene in the fault hypotheses taken into account in the design process.

The association into a component of its functional processing capability together with error detection mechanisms leads to the notion of **self-checking component**, either hardware or software; one of the important benefits of the self-checking component approach is the ability to a clear definition of the *error confinement areas*. The implementation of self-checking components through the multiple channel approach leads to the association of two components delivering the same service and of a comparison mechanism; this —widely used— form is often termed as *duplexing and comparing*[10].

Tolerance to temporary faults does not necessitate fault treatment, since error recovery should in this case directly remove the effects of the faults, which has itself vanished. However, distinguishing a permanent fault from a temporary fault is a complex task, since different fault classes may lead to very similar errors, and it can be said that declaring whether the cause of an error is a permanent or temporary fault is in fact subjective to some extent, including the fact that a fault may be declared as temporary if the fault diagnostics is unsuccessful.
In addition,

a) fault passivation is in fact a voluntary system change, which, as such, will lead to lowering the system available redundancy;
b) some faults, even permanent, have a likelihood of recurring which is low enough, or their consequences are bearable enough, so that fault treatment is not undertaken.

As a consequence, fault treatment generally does not immediately follow error processing; the delay may be determined by either logging a certain number of error occurences, or allowing a certain lapse of time during which the error(s) should not occur again. Another consequence, from the terminology viewpoint, is that temporary faults are often termed as **soft faults**, with respect to the fact that fault treatment is not necessary in their case; permanent faults are conversely termed as **hard**, or **solid, faults.**

Backward and forward error recovery are not exclusive: backward recovery may be attempted first; if the error persists, forward recovery may then be attempted. In forward recovery, it is necessary to *assess the damage* caused by the detected error, or by errors propagated before detection; damage assessment can —in principle— be ignored in the case of backward recovery, provided that the mechanisms enabling the transformation of the erroneous state into an error-free state have not been affected [And 81]. The tolerance to temporary faults may lead to a special form of forward recovery: the off-line (with respect to the current computation) restoration of the data corrupted by the temporary fault(s). In such a case, forward recovery is used in conjunction with possibly other forms of error processing; examples are provided by a) reading and rewriting the contents of a memory possessing an error correcting code for tolerating the effects of alpha particles, and b) in a triad of processors with a majority vote, restoring the data of one processor —affected by an electro-magnetic perturbation— from the data of the other two processors.
In fault masking, the systematic application of compensation ensures in itself that any error (provided of course it corresponds to the fault hypothesis of the design process) has been eliminated. However, this can at the same time correspond to a redundancy decrease which is not known. So, practical implementations of masking generally involve error detection, which enables compensation to be applied again, in order to check whether fault treatment has to be undertaken or not[11].
From the preceding discussion, it appears that the distinguished forms of error processing —backward and forward recovery, compensation— are in fact *primitives* which may be arranged in a wide variety of architectures.

Some previous definitions of compensation consider it within the context of interaction faults in distributed systems only, where one component provides to another component supplementary information intended to correct the effects of information it had previously sent (see e.g. [Ran 75]). The given definition clearly embodies more classical situations such as error correcting codes or majority voting[12].

The operational time overhead necessary for error processing may differ widely according to the adopted error processing form. Two extremes from this viewpoint are a) error detection and

10 Utilization of such self-checking components for fault tolerance may be coupled with both forms of error recovery:
- backward recovery when the self-checking components are processing different tasks;
- a —limiting— form of forward recovery when the self-checking components are processing identical tasks: switching from one component to another after error detection if the error has affected the component which was active from the user viewpoint.

11 Thanks to the masking effect, this second application of compensation may be performed within an acceptable delay, off-line with respect to the current computation which was in progress when the error became effective.

12 It is hoped that the essential idea has been captured, its applicability being a matter of definition of system boundaries.

backward recovery, and b) fault masking: the latter provides a time overhead which is much shorter than the former. In addition:

- in error detection and backward recovery, the time overhead is longer upon error occurrence than before: it is then related to the provision of recovery points, thus in fact to preparing for effective error processing;
- in fault masking, the time overhead is always the same.

This remark

a) is of high practical importance in that it often conditions the choice of the adopted fault tolerance strategy with respect to the user time granularity;

b) has introduced a relation between operational time overhead and structural redundancy; more generally, a redundant system always provides redundant behavior, incurring at least some operational time overhead; the time overhead may be small enough not to be perceived by the user, which means only that the *service* is not redundant; an extreme opposite form is "time redundancy" (redundant *behavior* obtained by repetition) which needs to be at least initialized by a structural redundancy, limited but existing; roughly speaking, the more the structural redundancy, the less the time overhead incurred.

Of importance is the signalling of a component failure to its users. This may be accounted for within the framework of *exceptions* [Cri 80, And 81]. *Exception handling* facilities provided in some languages may constitute a convenient way for implementing error recovery , especially in forward recovery for well defined and anticipated situations. However, the use of the term "exception", due to its origin of coping with exceptional situations —not only errors— should be carefully used in the framework of fault tolerance: it could appear as contradicting the view that fault tolerance has to be a natural attribute of computing systems, taken into consideration from the very initial design phases, and not an "exceptional" attribute.

3- On Maintenance

The frontier between fault treatment and corrective maintenance is relatively arbitrary; especially, corrective maintenance may be considered as an —ultimate— means of achieving fault tolerance. The given definitions were adopted for the ability to embody:

- on-line or off-line maintainable fault tolerant systems, as well as non fault tolerant systems
- preventive as well as corrective maintenance.

Especially, the faults whose removal is the aim of preventive maintenance can be a) physical faults having occurred since the last preventive maintenance actions, or b) design faults having led to effective errors in other similar systems.

It is noteworthy within the present context that the current discussions about the irrelevance of the use of the term "maintenance" when applied to software simply forget the etymology of the word: in the Middle Age, maintenance was designating the actions performed in order to keep an army in a state of giving battle, thus including the so-called "adaptative" and "perfective" forms of maintenance. The association of maintainance with repairing hardware is actually a (recent) deviation; associating "to maintain" with the notion of service would enable this etymological meaning to be revived, while at the same time removing the very source of discussion.

CONCLUSION

The contents of this paper are devoid of any "Table of Stone" pretension: the efforts undertaken are only worthwile in so far as they manage to embody as wide as possible a range of concepts and therefore those efforts have to keep abreast of technology. Naturally, the associated terminology effort is not an end in itself: words are only of interest in so far as they transmit ideas, subject them to criticism, and enable viewpoint to be shared.

The independence of the basic definitions with respect to any fault class should facilitate the bringing together of activities which are often considered as separate, such as:

- VLSI testing and software testing;
- hardware reliability (with respect to physical faults) and software reliability (with respect to design faults), to say nothing of hardware reliability with respect to design faults;
- computer system security, safety, and reliability.

ACKNOWLEDGEMENTS

What has been dicussed would not exist without the many discussions held with many colleagues. My thanks go to the members of the "Dependable Computing and Fault Tolerance" research group at LAAS, and to the members of the IFIP WG 10.4 "Reliable Computing and Fault Tolerance". Tom Anderson, Al Avizienis, Bill Carter, and Alain Costes deserve a special attention.

REFERENCES

And 81 T. Anderson, P.A. Lee, *Fault Tolerance — Principles and Prectice,* Prentice Hall, 1981.

Avi 78 A. Avizienis, "Fault tolerance, the survival attribute of digital systems", *Proceedings of the IEEE,* vol. 66, no. 10, Oct. 1978, pp. 1109-1125.

Avi 84 A. Avizienis, J.P.J. Kelly, "Fault tolerance by design diversity: concepts and experiments", *Computer,* vol. 17, no. 8, Aug. 1984, pp. 67-80.

Avi 86 A. Avizienis, J.C. Laprie, "Dependable computing: from concepts to design diversity", *Proceedings of the IEEE,* vol. 74, no. 5, May 1986, pp. 629-638.

Boe 79 B.W. Boehm, "Guidelines for verifying and validating software requirements and design specifications", in *Proc. EURO IFIP'79,* London, Sept. 1979, pp. 711-719.

Car 68 W.C. Carter, P.R. Schneider, "Design of dynamically checked computers", in *Proc. IFIP'68 Cong.,* Amsterdam, 1968, pp. 878-883.

Car 79 W.C. Carter, "Fault detection and recovery algorithms for fault-tolerant systems", in *Proc. EURO IFIP'79,* London, Sept. 1979, pp. 725-734.

Che 78 L. Chen, A. Avizienis, "N-version programming: a fault-tolerance approach to reliability of software operation", in *Proc. 8th IEEE Int. Symp. on Fault Tolerant Computing (FTCS-8),* Toulouse, France, June 1978, pp. 3-9.

Cri 80 F. Cristian, "Exception handling and software fault tolerance", in *Proc. 10th IEEE Int. Symp. on Fault Tolerant Computing (FTCS-10),* Kyoto, Japan, Oct. 1980, pp. 97-103; also in *IEEE Trans. on Computers,* vol. C-31, no. 6, June 1982, pp. 531-540.

Cri 85 F. Cristian, "Exceptions, failures and errors", *Technique et Science Informatiques,* vol. 4, no. 4, 1985, pp. 385-390; in French, English version avail. as IBM Reseach Report no. RJ 4130, Sept. 1983.

Elm 72 W.R. Elmendorf, "Fault-tolerant progtamming", in *Proc. 2nd IEEE Int. Symp. on Fault Tolerant Computing (FTCS-2),* Newton, Massachusetts, June 1972, pp. 79-83.

Gol 82 J. Goldberg, "A time for integration", in *Proc. 12th IEEE Int. Symp. on Fault Tolerant Computing (FTCS-12),* Santa Monica, CA, June 1982, p. 42.

Gra 78 J.N. Gray, "Notes on data base operating systems", in *Operating Systems,* Lecture Notes in Computer Science 105, Berlin: Springer-Verlag, 1978, pp. 394-479.

Hay 76 J.P. Hayes, "A graph model for fault-tolerant computing systems", *IEEE Trans. on Computers,* vol. C-25, no. 9, Sept.1979, pp.875-884.

Hos 60 J.E. Hosford, "Measures of dependability", *Operations Research,* vol. 8, no. 1, 1960, pp. 204-206.

IEE 82 IEEE Std. 729, *IEEE Standard Glossary of Software Engineering Terminology,* New York: IEEE, 1982.

Jes 77 D.C. Jessep, "Fault-tolerant computing, definition of terms", Report IEEE Computer Society P610/DI, Feb. 1977.

Lam 81 B.W. Lampson, "Atomic transactions", in *Distributed Systems — Architecture and implementation,* Lecture Notes in Computer Science 105, Berlin: Springer-Verlag, 1981, chap. 11.

Lap 82 J.C. Laprie, A. Costes, "Dependability: a unifying concept for reliable computing", in *Proc. 12h IEEE Int. Symp. on Fault Tolerant Computing (FTCS-12),* Santa Monica, CA, June 1982, pp. 18-21.

Lap 85 J.C. Laprie, "Dependable computing and fault tolerance: concepts and terminology", in *Proc. 15th IEEE Int. Symp. on Fault Tolerant Computing (FTCS-15),* Ann Arbor, Michigan, June 1985, pp. 2-11.

Pea 80 M. Pease, R. Shostack, L. Lamport, "Reaching agreement in the presence of faults", *Journal of ACM,* vol. 27, no. 2, Apr. 1980, pp. 228-234.

Ran 75 B. Randell, "System structure for software fault tolerance", *IEEE Trans. on Software Engineering,* vol. SE-1, no. 2, June 1975, pp. 220-232.

Ran 78 B. Randell, P.A. Lee, P.C. Treleaven, "Reliability issues in computer system design", *Computing Surveys,* vol. 10, no. 2, June 1978, pp. 123-165.

Sie 82 D.P. Siewiorek, R.S. Swarz, *The Theory and Practice of Reliable System Design,* Digital Press, 1982.

Tay 80 D.J. Taylor, D.E. Morgan, J.P. Black, "Redundancy in data structures: improving software fault-tolerance", *IEEE Trans. on Software Engineering,* vol. SE-6, no. 6, Nov. 1980, pp. 383-394.

Wen 78 J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostack, C.B. Weinstock, "SIFT: the design and analysis of a fault-tolerant computer for aircraft control", *Proceedings of the IEEE,* vol. 66, no. 10, Oct. 1978, pp. 1255-1268.

Zie 76 B.P. Ziegler, *Theory of modeling and simulation,* New York: John Wiley, 1976.

GLOSSARY

Warning: this glossary is provided as an aid for reading the paper. Do not consider it independently of the paper.

Availability	Measure of proper service delivery with respect to the alternation proper-improper service.
Coverage	Measure of the representativity of the situations to which a system is submitted during its validation with respect to the situations it will be confronted with during its operational life.
Dependability	Property of a computing system which allows reliance to be justifiably placed on the service it delivers.
Impairments to ~........	1) Undesired, but not unexpected, circumstances causing or resulting from un-dependability. 2) Faults, errors, and failures
Means for ~	1) Methods and techniques enabling a) to provide a system with the ability to deliver a service on which reliance can be placed, and b) to reach confidence in this ability. 2) ~ Procurement and ~ validation.
Measures of~.................	1) Attributes enabling the service quality resulting from the impairments and the means opposing to them to be appraised. 2) Reliability, availability, maintainability, safety.
~ Procurement	1) Methods and techniques intended to provide a system with the ability to deliver the specified service. 2) Fault avoidance and fault tolerance.
Un ~.......................................	Property of a computing system such that reliance cannot, or will not, any more be justifiably placed on the service it delivers.
~ Validation	1) Methods and techniques intended to reach confidence in a system's ability to deliver the specified service. 2) Fault removal and fault forecasting.
Design diversity	Approach for the production of systems providing identical services from independent designs and implementations.
Error	1) Part of system state which is liable to lead to failure. 2) Manifestation of a fault in a system.

Backward ~ Recovery	Form of error recovery where the erroneous state transformation consists of bringing the system back to an already occupied state.
~ Compensation	Form of error processing when erroneous state contains enough information to enable proper service delivery.
~ Detection	The action of identifying that a system sate is erroneous.
Detected ~	Error identified as such by a detection algorithm or mechanism, or noted by the user.
Forward ~ Recovery	Form of error recovery where the erroneous state transformation consists of finding a new, never occupied, state, or which was not occupied since the same error occurred.
Latent ~	Undetected error.
~ Processing	The actions taken in order to eliminate an error from a system.
~ Recovery	Form of error processing where an error-free state is substituted to an erroneous state.
Failure	1) Deviation of the delivered service from the specified service. 2) Manifestation of an error on the delivered service. 3) Transition from proper service delivery to improper service delivery.
Benign ~	Failure whose consequences are of the same order of magnitude as those of proper service delivery.
Catastrophic ~	Failure whose consequences are not commensurable with those of proper service delivery.
Fault	1) Adjudged or hypothesized cause of an error. 2) Error cause which is intended to be avoided or tolerated. 3) Consequence of the failure of a system which has interacted or is interacting with the considered system.
Active ~	Fault producing an error.
~ Avoidance	Methods and techniques aimed at preventing fault introduction.
Design ~	Human-made fault.
~ Diagnostics	The action of determining the cause of an error in location and nature.
Dormant ~	Fault not activated by the computation process.
External ~	Fault resulting from the system's interference with its environment.
Hard ~	Synonimous of permanent fault.
~ Forecasting	Methods and techniques aimed at estimating the presence, the creation, and the consequences of faults.
Human-made ~	Consequence of human imperfection.
Intermittent ~	Temporary internal fault.
Internal ~	Fault inside a system.
~ Masking	Result of applying error compensation systematically.
~ Passivation	The actions taken in order that a fault cannot be activated.
Permanent ~	Irreversible structural state.
Physical ~	Fault resulting from physico-chemical disorders.
~ Removal	Methods and techniques aimed at minimizing the presence of faults.
Soft ~	Synonimous of temporary fault.
~ Tolerance	Methods and techniques aimed at providing service complying with the specification in spite of faults.
Temporary ~	Fault whose action is not longer than a certain maximum time.
~ Treatment	The actions taken in order to remove a faulty component from the computation process.
Transient ~	Temporary physical external fault.
Maintainability	1) Measure of continuous improper service delivery. 2) Measure of the time to restoration.
Maintenance	Actions undertaken in order to remove faults.
Corrective ~	Maintenance aimed at removing faults which have produced detected errors.
Preventive ~	Maintenance aimed at removing faults before they are activated.

Recovery point	Point in time during the execution of a process for which the then current state may subsequently need to be restored.
Reliability	1) Measure of continuous proper service delivery. 2) Measure of the time to failure.
Safety	1) Measure of continuous delivery of either proper service or improper service after benign failure. 2) Measure of the time to catastrophic failure.
Self-checking component	Component comprising error detection mechanisms associated to its functional part.
Service	System behavior as perceived by the system user.
Proper ~	Service delivered as specified.
~ Restoration	Transition from improper to proper service delivery.
Improper ~	Service delivered differently from specified.
~ Specification	Agreed description of the expected service.
System	1) Entity having interacted, interacting , or able to interact with other entities. 2) Set of components bound together in order to interact.
Atomic ~	System whose internal structure cannot be discerned, or is not of interest and can be ignored.
~ Behavior	What a system does.
~ Component	Another system.
~ Environment	The other systems interacting or interfering with the considered system.
~ Structure	What enables a system to do what it does.

THE APPLICATION OF EMULATION TECHNIQUES IN THE ANALYSIS OF HIGHLY RELIABLE, GUIDANCE AND CONTROL COMPUTER SYSTEMS

Gerard E. Migneault
Langley Research Center
Hampton, Virginia, USA

Summary

Emulation techniques can be a solution to a difficulty that arises in the analysis of the reliability of highly reliable guidance and control computer systems for future commercial aircraft.

This paper first describes the difficulty, viz., the lack of credibility of reliability estimates obtained by analytical modelling techniques. The difficulty is shown to be an unavoidable consequence of: (1) a reliability requirement so demanding as to make system evaluation by use testing infeasible, (2) a complex system design technique, fault tolerance, (3) system reliability dominated by errors due to flaws in the system definition, and (4) elaborate analytical modelling techniques whose precision outputs are quite sensitive to errors of approximation in their input data. Use of emulation techniques for "pseudo-testing" systems to evaluate bounds on the parameter values needed for the analytical techniques is then discussed. Finally, several examples of the application of emulation techniques are described.

Introduction

About a decade ago, designs for fault tolerant computer systems began to surface in anticipation of the guidance and control computing needs of future civil aircraft (e.g., [Bjurman, B. E. et al., 1976], [Hopkins, A. L. et al., 1978], [Wensley, J. H. et al., 1978]). The reasons were clear and remain so. On the one hand, demand for safety imposes an elevated, acceptable level of reliability on any system whose failure can cause fatalities; on the other hand, there is uncertainty about the quantitive relationship between subtle malfunctions of an aircraft's guidance and control computing system and the probability of catastrophic consequences involving the aircraft. As a result, any specification of a minimally acceptable level of reliability for an aircraft's computing system remains problematical and so, to be generally acceptable, must be conservatively elevated. Credibly conservative levels, however, preclude the use of conventional (i.e., fault intolerant) systems because of limited levels of reliability of available components.

In early studies, for example, requirements for an acceptable level of reliability of the systems and associated components were inferred from the expression "extremely improbable" which appeared in regulatory documentation pertaining to safety in commercial transport aircraft (FAA, 1970). The following, variously worded, informal statements indicate the range of interpretations of the expression in several studies commissioned by NASA:

> "Thus we have a reliability requirement of 10^{-8} per hour of operation for a level 1 or level 2 function¹ with no internal or external backup ..." [Ratner, R. S. et al., 1973]
>
> "... a number less than or equal to $1x10^{-9}$ has been imposed ... to represent the probability of an event designated as extremely improbable. ... Loss of the CCV/FBW² function, given a fault-free system at dispatch, shall be extremely improbable." [Bjurman, B. E. et al., 1976]
>
> "... the computer's failure rate will be designed below 10^{-9} failures per hour in flights of up to ten hours duration, with a preferred goal of 10^{-10} failures per hour." [Smith, T. B. et al., 1978]
>
> "... the extrapolated failure of the design in context with production system application shall not exceed 10^{-9} computer-related system failures in flights up to ten hours."(sic) [NASA, 1978]

---

¹/ Levels pertain to criticality of functions, levels 1 and 2 being most critical.

²/ CCV/FBW = control configured vehicle / fly by wire.

Current regulatory agency documentation is somewhat more explicit in associating "extremely improbable" with the probability value 10^{-9}, but otherwise there has been little change [FAA, 1982]. And one can expect that there will not be much change in the future. Thus, the following, informal statement, which is more or less an average of the cited interpretations, will likely remain approximately the requirement for reliability for a system crucial to flight:

> the probability that the system will fail during a flight of up to ten hours duration will be less than approximately 10^{-9}.

The understanding is implicit that the requirement applies anew to each flight and that, therefore, a renewal process is being considered. (In military, combat aircraft, requirements for reliability of computing systems hover about the value $1-10^{-7}$ for a mission -- still beyond the sure reach of computing systems that are intolerant of faults).

If one ignores failures due to causes external to systems or to inadequately or incorrectly designed and implemented systems, one can determine that, in order to satisfy the reliability requirement above, a computer system constructed of devices (that may in turn be constructed of more basic components) with independent failure distributions and constant failure rates would require, if it were intolerant of the failure of any of its constituent devices, a mean time to failure (MTTF) of approximately ten billion (10^{10}) hours <u>for the least reliable of the devices</u>. Such a system is unlikely to see the light of day in the near future, to say the least, since realistic, available devices such as processors, memories, etc., from which systems can be constructed, do not have such lengthy MTTFs; values in the range from 10^2 to 10^5 are more reasonable. Consequently, computer systems intended to satisfy the reliability requirement have been designed to <u>tolerate</u> failures.

<u>Consequences of Fault Tolerance</u>

The inherent characteristics of fault tolerance for achieving high reliability give rise to a need to examine explicitly the implications of a failure mode, failure resulting from a latent fault, conventionally handled implicitly by testing actual system.

A first, rather obvious characteristic of a fault tolerant system is the redundancy of its components -- at the very least when in an initial condition free of failed components. In the case of systems with requirements for reliability stated in terms of the first few hours or a small fraction of expected equipment lifetimes, renewal activities are needed to ensure that the systems are in a condition perceived to be fault-free before another period of use is begun. This fact, combined with the MTTFs of realistic, avionics devices ensures that considerable amounts of repair activity will be needed -- to return systems to the fault-free, initial condition needed to fulfill the assumptions underlying the reliability estimates. Thus, the characteristic further suggests, other things unchanging, that the more "multifunction" the constituent devices are, the more efficient the systems are in terms of total equipment used and maintained. That is to say, the strategy used to gain the desired reliability goal gives rise to another goal which cannot be ignored, an economic pressure for designs utilizing multifunction devices such as microprocessors with software. In striving toward the economic goal, however, a cost is incurred in a different coin, i.e., greater complexity in the synthesis, logic, and analysis of systems with parallel and intersecting signal and data paths and time-shared use of resources and algorithms.

Another, necessary characteristic of a fault tolerant system is its possession of an agent or mechanism that embodies the intelligence by which redundancy is put to effective use for detecting and nullifying the erroneous outputs of devices or components. This characteristic may take on a passive form when some fortuitouus property of nature is present (e.g., parallel rather than series wiring of Christmas tree lamps guards against an open circuit failure caused by one defective lamp), or, as appears more likely to be necessary in complex systems, an active form when still more devices and logic must be added to a redundant system to act as error detectors and nullifiers. Of course, a price is paid again in increased complexity. Moreover, there is a new failure mode that did not exist before, viz., failure due to a design flaw in the mechanism. Such a flaw is more pernicious than a simple, latent fault lying dormant in the other part of the system until an unlikely situation awakens it; a flaw in the mechanism for reacting to errors necessarily implies a multi-failure event which has not been foreseen by the designers of the system. Analysis of the reliability of a fault tolerant system rests on a precarious perch.

There is a notion which merits a few words at this point for it is about here that it occasionally arises. The notion is that the requirement for reliability is unnecessarily stringent, as witnessed by the ten billion (10^{10}) hour MTTF previously cited. (Indeed, some critics consider the goal of ultra reliability to be infeasible at best, and the resulting schemes for fault tolerant computers as "redundancy 'run amok'" [Goldberg, H., 1981, pp 220-222]). Ten billion hours, however, was the value given for the MTTF (easily computed since it equals the reciprocal of the constant failure rate) of a fault intolerant system and is inappropriate as a figure of merit for a fault tolerant system of equivalent reliability at an extremely early stage (i.e., ten hours) of its expected operation. To understand this, observe the two curves in Figure 1; the curves represent the (failure) probability density distributions of two systems having the same mean (i.e., the same MTTF). Density A corresponds to a fault intolerant system; density B, to a fault tolerant system (with redundancy in the form of simple parallelism). If one supposes that the system of density A satisfies the requirement for reliability at the required early time (i.e., the area for the first ten hours under the density A curve is 10^{-9}), then it is abundantly clear that the system of density B is excessively reliable -- and unnecessarily costly. Clearly, the parameters of the system of density B can be relaxed and the required reliability at ten hours still achieved; but then, its MTTF will be much less than that of density A.

A better figure of merit for a fault tolerant system is provided by the MTTF of systems composed of several r-out-of-n subsystems (i.e., n parallel, identical devices of which r must be operating for the subsystem to be operating) in series. And conveniently, a system consisting of a single r-out-of-n subsystem serves as a reasonable upper-bound estimate of the MTTF of a fault tolerant system when the representative constituent device chosen is the fault tolerant system's "worst" (i.e., the device type with the greatest MTTF in the set of constituent devices whose functions cannot be performed by any combination of the other device types of the system; a processor would be most likely in this set). Assuming, as before, that constituent devices have independent failure distributions and constant failure rates, one can show that an r-out-of-n system has a MTTF not very much different from that of its constituent device, and quite likely less because of factors accounted for by "coverage". Figure 2 contains a simplified behavior model of an r-out-of-n system. Each state corresponds to a set of possible configurations having a stated number of operating constituent devices. The transition rate out of a state is (approximately, for small λ) the appropriate multiple of the constant failure rate, λ, of one device. Since, given the occurrence of a component failure, a successful transition to another operating state of less redundancy is problematical, so-called "coverage" parameters, C_j, are included. They are conditional probabilities of successful transition given a failure; unsuccessful transitions, with conditional probabilities $(1 - C_j)$, are assumed to cause immediate system failure. Usually the coverage parameters are associated with systems having active recovery processes, but they are also applicable to passive mechanisms as long as there are transitions which can go awry among distinguishable, operating states. No distinction is made here. Recognizing this model and its assumptions as a Markov process, one can easily develop the appropriate differential equations for the stochastic process [Feller, W., 1966] and determine in a straightforward manner that the probability of system failure by time t is represented by the expression

$$1 - e^{-n\lambda t} \sum_{j=0}^{n-r} a_j \binom{n}{j}(e^{\lambda t} - 1)^j$$

where $a_0 = 1$ and $a_j = \Pi_{i=0}^{j} C_j$ for $j = 1,2,\ldots,(n-r)$.

Ratios of system MTTF to constituent device MTTF are tabulated for various combinations of values of r, n, and C_j in Tables 1 and 2. In Table 1, $C_j = 1$ for all j, implying that coverage is perfect. Although the ratios are independent of the constituent device's failure rate (or equivalently, MTTF), not all combinations of r and n are useful, given a specific device failure rate, when the 10^{-9} requirement is considered. For instance, a device with MTTF less than 10^p hours could be used to construct systems with r and n values corresponding to zones p-1 and lower marked on the Table, but not zones p or higher. More specifically a device with MTTF of five thousand ($5x10^3$) hours would not be used to construct systems of zones 4 and 5. In Table 2, $C_1 = 0.9$ and $C_j = 0.1$ for all $j \neq 1$, which is excessively poor coverage since systems are all in zones 9 or higher. In all cases in both Tables the ratios do not differ from 1 by an order of magnitude. Hence, to the extent that fault tolerant systems are represented by r-out-of-n systems, a simple and reasonable approximation to the MTTFs of such systems appears to be simply the MTTF of the "worst" device type, a far cry from the ten billion (10^{10}) hour value.

However, having identified a better approximation to MTTF for fault tolerant systems, it is well to note that, in the application of interest, the systems will be effectively renewed every ten hours or so. Hence MTTF, in the conventional sense of an unrenewed system used until system failure, as computed above, is still not descriptive enough of a system's use. In order to consider the relationship of the reliability requirement to safety, it is more meaningful to estimate the probability that one or more system failures, which will be euphemistically called emergency situations, will occur during the lifetime of a fleet of aircraft with realistic policies for renewal. Therefore, assuming (1) systems meeting the 10^{-9} requirement when all failure modes are considered, (2) system renewal after every ten hours of operation, and (3) a fleet of two thousand ($2x10^3$) aircraft each with a lifetime of sixty thousand ($6x10^4$) hours, the probability is approximately 0.01 that one or more emergency situations will occur because of a computer system. It is a matter of judgment, no doubt tempered by economics, whether or not such a risk to safety is acceptable. Indeed, the estimate does not consider latent failures, i.e., conditions where physical defects have occurred but have not yet contributed to a data error because the failed components have not been party to a computation. Such a failure mode could be modelled as an aging effect on the systems -- despite periodic renewals. Hence, the inference must be made that the value 0.01 above is optimistic. And notice that the computation has not included any manner of considering the effect of increased complexity as $\underline{r}$ and $\underline{n}$ increase.

The increased complexity, while ostensibly reducing the incidence of system failures resulting from component and device failures, is ironically a major source of residual, definitional flaws in systems. The term "definitional flaw" is adopted here to denote an inadvertent system design which, when the system is in some particular condition with some unexpected data and regardless of the presence or absence of conventional component failures or anomalous environments, produces undesirable results which could have been avoided by another, proper design; the term includes design errors, specification errors or inadequacies, missing requirements, etc. It matters not whether the flaw is in hardware or software or is the result of the correct implementation of an erroneous or incomplete specification. The root cause is human error. Of course, one expects the incidence of such flaws to increase with growth in complexity. There is a quite large pool of practical experience with such a failure mode -- everyone's 'bêtes noires', the software bugs found in operational software systems -- which indicates strongly that the failure mode must be included, in some fashion, in the analysis of the reliability of complex systems. Yet, in the avionics applications of interest, the level of system reliability required effectively precludes the use of thorough, lifetime or use testing of actual systems to determine with acceptable confidence (i.e., confidence quantified in a statistical sense) that the probability of system failure due to residual, definitional flaws is compatible with the reliability goals and requirement. As a consequence, more analytical methods must be developed and relied upon to address total system (i.e., logic, largely software, and hardware) reliability (e.g., [Costes, A. et al., 1978]) -- but with acceptable credibility.

Addressing Definitional Flaws

Analogous to the techniques of hardware redundancy, there are techniques for designing systems with dissimilar redundancy (i.e., with components that are functionally redundant but implemented dissimilarly either in their hardware or software or both), and in the past decade they have become popular subjects of study and development -- to the point of becoming textbook material [Anderson, T. and Lee, P., 1981] and practical systems in aircraft (e.g., in the AIRBUS A310 [Martin, D.J., 1982] and in the Boeing 737-300 [Yount, L. J., 1984]). Clearly, dissimilar redundancy is intended to reduce the incidence of system failures attributable to residual, definitional flaws. The schemes apply equally to flaws in hardware design and software design, but they have been studied more with software in mind. Hence, software is used here to represent dissimilar redundancy effects in general. There are two, widely known, basic, software fault tolerance schemes (viz., N-Version Programming and Recovery Blocks with Acceptance Test), and one or the other is the root of almost every scheme, which is either a variant or an extension, proposed in the literature or put into practice. While the early schemes were oriented to a sequential algorithmic process, the later variants and extensions have considered the use of parallel, computing capabilities. Thus, today there is essentially a parallel of dissimilar redundancy techniques to the techniques of hardware redundancy.

Fault tolerant software lends itself to an especially simple behavior model, as shown in Figure 3(a), on the assumption that successful recovery from a data error caused by a fault in the software (or, similarly, by a definitional flaw in hardwired logic in the case of hardware) implies no degradation to a state representing a system with less capability or less redundancy. The rational for the assumption is that the flaw responsible for the data error was always present in the system; it had simply not been activated before, so to speak. Thus, if the system, by virtue of its fault tolerance characteristics, survives the data error, it can be expected to function as before (i.e., there will only be another data error when an another apparently unusual coincidence of state and input conditions occurs). (As an aside, experiments using the emulation techniquė are suggested by this observation -- experiments to determine whether or not software data errors might not better be modelled as error bursts). Figure 3(b) is a still simpler representation of the same failure and recovery process. Again for the sake of simplicity, software is assumed to have a constant failure rate, μ, and fault tolerant software is assumed to have an aggregate recovery parameter, k, analogous to the coverage parameters of the r-out-of-n hardware model. Immediate system failure is assumed to be the result of lack of successful recovery. No further elaboration of a software model is attempted. Indeed, short of the argument for adopting either a conventional or a Bayesian probabilistic model, there exists no justification, and certainly no empirical evidence, for selecting any other, neither more general nor more particular, model of system failure due to software, let alone due to the more general condition of residual, definitional flaws.

Reliability Analysis by Mathematical Modelling: How Credible?

The r-out-of-n model of Figure 2 and the software model of Figure 3 suffice, however, to show the difficulty, when lifetime testing of actual systems is not feasible, of establishing with acceptable confidence (in the statistical sense) that systems designed to satisfy the 10^{-9} requirement do achieve the reliability goal. In Figure 4, the two models are combined to represent simply a system subject to and tolerant of both hardware component failures and errors due to residual, definitional flaws (represented here by the software). An additional assumption is made to keep the illustration simple, viz., that the software and hardware behave independently. Clearly, that is a simplification and it is possible to add more complexity to the model, but, as stated before, there is no empirical evidence to justify selecting any particular model in preference to another. Also, and more to the point here, the conclusion below is not appreciably modified. Thus, again recognizing the model and its assumptions as a Markov process, the probability of system failure is computed to be

$$1 - e^{-(n\lambda+\mu[1-k])t} \sum_{j=0}^{n-r} a_j \binom{n}{j} (e^{\lambda t} - 1)^j$$

where a_0 and a_j are as before.

For a typical (and optimistic) value for λ ($\approx 10^{-4}$ failures per hour in a hardware component), typical values for n ($\approx$ 3 to 5 components in parallel), and the required value for t (= 10 hours), bounds on C_1, C_2, and $\mu(1-k)$ required for the system to satisfy the 10^{-9} requirement are calculated to be as follows:

$$0.999999 \le C_1 \le 1.0$$

$$0.9999 \le C_2 \le 1.0$$

$$\mu(1-k) \le 10^{-10}$$

Note that there is in general a coupling between physical and logical redundancy techniques and guidance and control schemes -- e.g., the choice of control laws is constrained to those capable of tolerating time lost in reconfiguration, etc. The numbers above leave little margin for error in designing systems to satisfy the 10^{-9} requirement. Refinement of the model cannot eliminate the difficulty in estimating precisely the reliability of such systems; it can only transform it into an equivalently demanding knowledge of a different set of parameters, for the systems must still achieve the same aggregate behavior as above.

More Complex Models

In the process of investigating fault tolerant systems (previously, principally studies of hardware), numerous models have been developed for analyzing the reliability of such systems. Studies have also been undertaken into models to relate the system failure modes to time-variable computational and performance requirements, thus attaching the reliability of a system more tightly to its applications [Meyer, J., 1977], [Beaudry, M. D., 1978], [Castillo, X. and Siewiorek, D., 1982], [Iyer, R. and

Rossetti, D., 1982]. And over the years many model evaluation schemes have been "computerized" to serve as more or less general purpose tools for the convenient analysis, in the architectural design stage, of systems composed of complex arrangements of elements, e.g., CAST [Cohn, R. B. et al., 1974], CARE II [Stiffler, J. J., 1974], CARSRA [Bjurman, B. E. et al., 1976], ARIES [Ng, Y., 1976], HARP [Trivedi, K. et al., 1984], SURE [Butler, R. W., 1984], and CARE III [Bavuso, S. and Peterson, P., 1985]. Although they consider details of systems behavior such as recovery (detection, isolation, reconfiguration) strategies, sparing (active, stand-by, switching) strategies, transient and intermittent fault (duration, periodicity, leakage) modes, functional dependence among devices, nonexponential failure distributions, etc., the models that the various "computerized" evaluation schemes handle are still constructed from parametric descriptions of aggregate system, subsystem and (or) device behavior in order to make use of mathematical techniques applicable to idealized stochastic processes and to reach levels of reasonably efficient computation. Hence, all the models contain parameters whose values need to be assumed or known, by some other means, in order to precisely represent any and each particular system design of interest. The credibility of any reliability assessment gotten by the use of these mathematical techniques obviously hangs upon the inaccuracies and imprecisions of the models, their forms and the values of the parameters. But knowledge of these inaccuracies and imprecisions can only come from empirical observations, i.e., testing and experimentation.

Lifetime or use testing of fault tolerant systems appears infeasible if the systems approach their reliability goals; however, selective testing and experimentation can be performed to obtain knowledge about specific aspects (e.g., values of parameters) of the mathematical models. The physical and statistical nature and number of flaws, faults, and failures, however, still make the use of actual systems impractical. Surrogates, preferably abstract, into which faults and flaws can be readily introduced are needed -- but they must be able to faithfully reproduce the behavior of the particular, actual systems. Digital simulations which mimic the working of the internal, logical structures of the systems being studied are such surrogates.

Digital Simulation

The word "simulation" is usually used to denote all manner of techniques for, among other purposes, analyzing the behavior of objects and their environments by means of implementation and manipulation of more conveniently malleable surrogates, but here the word is limited to mean the use of computer systems as surrogates -- at whatever level of abstraction found meaningful to an application. The concept of system is stressed because usefulness of a simulation scheme depends upon both software and hardware -- a characteristic more effectively utilized by emulation. For example, consider even the high level reliability analysis programs previously mentioned -- CAST, etc. Although they are essentially simulation schemes which are normally discussed without regard to host computer hardware, in any actual application, the hardware of the host computer will be an important constraint upon the amount of detail which it will be feasible to consider with the programs.

Digital simulation at the level of gate logic has been discussed for some time in the literature on computers and considered as a tool for design and fault (signature) analyses of digital logic circuits at levels of detail ranging from simple (e.g., assuming gates to have only two possible output values) to complex (e.g., allowing undefined values of gate outputs and various timing anomalies) [Szygenda, S. and Thompson, E., 1976]. For the analysis of circuits the sizes of microprocessors, memories, and larger, simulation techniques at a more aggregate, functional behavior level have been used (e.g., [Menon, P. and Chappell, S., 1977] and [Armstrong, J. R. et al, 1985]) as the gate level simulation costs become prohibitive when compared to perceived benefits.

However, for the purposes of reliability analysis of fault tolerant systems, gate level simulation warrants considerable cost in view of the conclusion to be drawn from the preceding paragraphs that, at the levels of reliability of interest, the probability of failure of such systems is less dependent upon the mode of failure resulting from depletion of redundant resources than it is upon the less well understood and questionably modeled modes considered under the terms "coverage" and "definitional flaws". A similar conclusion to the effect "that the introduction of a redundancy at the hardware level increases the relative influence of software faults" is made elsewhere (Costes, A. et al., 1978). Unfortunately, while the costs could be borne, in light of the benefits, gate level simulation is not always a feasible technique for application to questions involving chance events and repeated trials because it is time consuming -- orders of magnitude slower than likely target systems.

Emulation

In ordinary use, the word "emulation" means an endeavor to equal or excel; in the present context, it is reserved for a particular technique of implementing simulation possible when the hardware of the host (computer) is dedicated to the task (e.g., hardwired or microprogrammed). The technique is most often associated with hardware implementations of logic simulators for computer-aided design applications [Blank, T., 1984. Microprogramming is significant because it provides an alternative to the creation of special hardware for emulation purposes. In general, microprogramming allows a final definition of a host computer's "apparent" instruction set to be postponed until after the definition of the host's hardwired logic is completed, and it does this with an acceptably small risk that the hardwired logic will need redesign. This happens because a "real" instruction set is defined by the hardwired logic, is at a quite primitive level, and is tailored especially for executing algorithms which, in turn, become operational definitions of less primitive operations -- the "apparent" instruction set. It may be said that a computer defined by an "apparent" instruction set does not really exist; it is "emulated" by microprogrammable hardware by means of microcoded algorithms. Admittedly, variations in efficiency of variant microcode operations vis-a-vis various "apparent" instruction sets may exist, but they can be ignored for the present purpose. What is notable is that, given reasonable care not to mismatch host and target computers, microprogrammable computers can perform in the role of an "apparent" computer approximately as efficiently as would either a hardwired version of the "apparent" computer or a hardwired emulation algorithm. Of course, the existence of parallel processing capability either in the emulated or the host hardware can alter the relative speed of an emulation. Nevertheless, by the use of microprogramming, emulation occurs at a level of detail which permits software implemented for another, "apparent", target computer to be executed directly by a host computer. That is, no modification of the target software is needed to make it compatible with the host computer, and no special software on the host computer needs to be generated (more importantly, no simulation program in an "apparent" instruction set on the host to interpret the instructions of the target software and mimic the target computer) as would be needed on a non-microprogrammable computer.

Addition of diagnostic, control functions to an emulation capability, as an add-on to hardwired emulation logic or in the microcode of a microprogrammable host computer, permits the host to act not only as a surrogate but also as a device for observing and recording (and possibly analyzing) target software performance in an ostensibly natural environment. Such "diagnostic emulation" has been used for the development and maintenance of special software systems and is, seemingly, "emulation" in the dictionary sense. As might be expected efficient use of such a diagnostic system requires support capabilities for readily modifying microcoded algorithms defining target computers. What has been less well considered is the fact that such capabilities can be extended to permit analysis not only of software but also of systems (i.e., software and hardware) -- and not only as they are intended to be but also as they are not. By emulating target computer systems in sufficiently fine detail, combinations of failures in individual components, anomalous data, and definitional flaws can be introduced and their effects at the system level observed rather than assumed. Thus, emulation provides a conveniently manipulated failure effects analysis tool.

In addition, with automated diagnostic and system and environment controls, emulation can be used to generate repeated trials of "emulated" systems from which failure ratios and histograms can be tabulated for analysis -- hence, aggregate behavior models verified and parameter values estimated with some measure of confidence (in a statistical sense). Clearly, assumptions about the manners and rates of occurrence of failures and flaws must still be made in order to introduce these last into the emulations. However, while the credibility of precise assumptions will still be questionable, credibly pessimistic assumptions could be used to demonstrate that particular fault tolerant system designs exceed their reliability requirement.

Examples of the use of Emulation

As examples of the application of emulation to analyses of the reliability of fault tolerant systems, consider the following two studies -- one long completed, the other underway. In addition to their obvious purposes of analysis, the studies were done to provide experience in the use of an emulation capability for the AIRLAB facility at NASA's Langley Research Center. The implemented emulation technique was proposed in 1976 precisely for providing abstract surrogates credibly representative of the internal structures of fault tolerant, digital systems of the types to be used in avionics and guidance and control systems for aircraft. There are two current implementations of the algorithms that comprise the technique: one is as a code running on a dedicated, horizontally microprogrammable computer; the other runs on a general purpose computer. (There are, of course, other manners of implementing the technique, as there are other algorithmic processes for simulating the behavior at the logic level of digital, computer systems -- e.g., GLOSS [M^{c}Gough, J., 1983], also developed for use in the AIRLAB). The microprogrammed technique may be summarized as follows:

1- It is a hybrid scheme. That is, although the nominal level of detail is the gate logic level, the scheme facilitates the representation of different parts of a digital system at different levels of detail -- usually, for computational efficiency, at a less detailed and more functional level, but also, if needed, at a more detailed level than even the gate logic level.

2- It is a generic scheme in the sense that it uses the same algorithm for all emulated networks and in contrast to embedded network schemes that compile the system definitions into a computer program. Embedded network schemes presumably need a compilation for each separate network and, therefore, additional validation; thus, they appear more suitable to simple verification and validation of software -- where there is no consideration of failures in the underlying hardware.

3- It is a quasi event-driven scheme. That is, emulated time advances by fits and bounds to the next dynamically scheduled event(s); no host computer time (or resource) is consumed emulating inactive systems -- but at a cost of maintaining a dynamic schedule of events.

4- It is a unit propagation delay scheme -- for the most part -- in its handling of gates. In its normal mode of use all gates are presumed to share the same propagation delay. However, varied delays can be accomodated by specially definable means.

5- It is mainly a binary valued scheme, since its principal data structure accomodates only TRUE and FALSE values for the outputs of its nominal, primitive devices. However, more complex, multi-state behavior can be accomodated, at a cost in running time, again by specially definable means.

6- Finally, it is a sequential processing scheme because of the limitations of the host hardware. The algorithmic scheme does, however, contain latent parallelism corresponding to concurrency among events in the system being emulated; advantage, therefore, could be gained from host hardware which could accomodate the parallelism.

The First Example: The first effort, of limited scale, was undertaken in order to determine whether or not an emulation scheme could be devised which would be sufficiently efficient to support analyses of target systems of meaningful sizes and complexities, and to demonstrate that such a scheme could be implemented in a manner convenient for analysis purposes by users not well versed, it at all, in the emulation scheme itself. The study was performed on a large, general purpose computer whose underlying micro-code was sacrosanct. For that reason, emulation was really simulated. This last level of complication can be accounted for by introducing a time scale factor and otherwise ignored here. The analysis performed was a study of the efficacy of five (5) particular algorithms, each with a different instruction mix, as detectors of component "stuck-at" faults (i.e., latent failures) in a hypothetical target computer. The analysis is documented in detail in [Nagel, P., 1978].

The hypothetical, target computer was originally generated (i.e., defined at the gate logic level) as a vehicle for checking out the original (and modified) versions of the emulation algorithms, and for demonstrating the ability of support software, a hardware description language translator and meta-assembler for regenerating target software, to respond semi-automatically to hardware design changes. The hypothetical computer had a memory of 8192, 16 bit wide words, a CPU with a count of approximately 2000 gate equivalents, and a single input-output register/port. The logic was arbitrarily assigned to four (4) hypothetical chips: a "clock" chip, an "adder" chip, an "op-decode" chip, and a miscellaneous odds and ends chip. The instruction set contained about a dozen basic instructions.

The emulated system trials were simple. The five algorithms, ranging in length from about a dozen instructions to several hundreds, were repeatedly executed, with randomly selected initial data, and randomly selected faults of random components. Distributions of time from fault occurrence to fault detection (i.e., fault latency duration) were measured. Two analyses of the sort that would be of interest in studies of fault tolerant systems were made. First, the observed distributions were fitted against commonly used mathematical models, e.g., exponentials, as would be done in order to determine models and parameter values for use in reliability analysis programs. Although the results were not initially considered significant, owing to the fanciful nature of the emulated system, they were later corroborated by an analogous but more thorough study using a real computer [M^{c}Gough, J. and Swern, F., 1981]. That the distributions each exhibited different nonzero probabilities of never detecting the faults was predictable, but only an experiment of this nature could determine the differences in magnitude. Secondly, a search was made for correlations among the distinguishable characteristics of the algorithms and the distributions. The only significant correlation found was between instruction mix and detection probability. Here too, because of the nature of the target system, the magnitudes of the correlations were not considered definitive. But the concept is nevertheless a useful one -- for determining which characteristics should be avoided in algorithms whose function is to reconfigure a system after a failure has been detected, or which characteristics should be used in those algorithms whose function is to detect latent failures. The latter aspect is particularly applicable to the purpose of the second example below.

The Second Example: Currently, the emulation system is being used to examine the self-diagnostic characteristics of the Communicator/Interstage (C/I) of a particular Fault Tolerant Processor (FTP) (described in [Smith, T. B., 1984]) proposed for use in the Advanced Information Processing System (AIPS) being developed for NASA's Office of Aeronautics and Space Technology. In the study [NASA, 1985], the target system consists of the FTP's four C/Is, which interface with each other and with a number of processors (four at the time of writing) that compute <u>redundantly</u>. Among other tasks, the C/Is each vote the redundant outputs, exchange, and pass on (return) the voted output -- thus, hopefully, ensuring that the processors work upon the same, mutually agreed upon quantities at the next iteration. Under control of the processors, diagnostics can be performed in search of latent, hardware faults in the C/Is, whose logic consists of about four thousand ($4x10^3$) gate equivalents and a small amount of read only memory.

In the study, various faults will be inserted into the C/Is to discover what proportion will be detected by the preplanned diagnostic test vectors. Two types of outcomes are anticipated. First, the study will verify that the set of test vectors will detect all inserted faults. Should this not prove to be so, the set of test vectors will be expanded and the study repeated until the statement is true. At the same time, and as a secondary benefit, "superfluous" test vectors will be identified -- for the obvious reason of making the diagnostic less time consuming, more efficient, etc.; such a result was achieved in a similar study of the self-diagnostic code for an avionics processor [M^{c}Gough, J. and Swern, F, 1983]. Secondly, distributions of the times to discovery of the faults will be developed, information of direct relevance in the development of appropriate mathematical models for analyzing the reliability of the FTP.

<u>Conclusion: Reprise</u>

A case has been made for the use of emulation techniques as a needed and useful adjunct to mathematical models for the reliability analysis of highly reliable avionics and guidance and control computer systems. Given the lack of lifetime testing of actual systems, without, at the very least, the use of such <u>surrogate</u> analysis tools the trust to be placed in fault tolerant systems in life-critical situations requiring high reliability must remain dubious.

<u>REFERENCES</u>

Anderson, T. and Lee, P. A., <u>FAULT TOLERANCE Principles and Practice</u>, Prentice/Hall International, Inc., Englewood Cliffs, New Jersey, 1981

Armstrong, J. R., Tront, J. G., and Li, K. W., "Modeling and Simulation of the Effects of Internal Transient Upsets on Microprocessors", in <u>TRANSACTIONS of The Society for Computer Simulation</u>, Vol 2, No 1, May 1985, pp 73-93

Bavuso, S. J. and Petersen, P. L., <u>CARE III Model Overview and User's Guide</u>, 1st revision, NASA TM 86404, Langley Research Center, Hampton, Virginia, April 1985

Beaudry, M. D., "Performance-Related Reliability Measures for Computing Systems", in <u>IEEE Transactions on Computers</u>, Vol C-27, No 6, June 1978, pp 540-7

Bjurman, B. E., Jenkins, G. M., Masreliez, C. J., McClellan, K. L., and Templeman, J. E., Boeing Commercial Airplane Company, <u>Airborn Advanced Reconfigurable Computer System</u>, NASA Contractor Report 145024, Langley Research Center, Hampton, Virginia, August 1976

Blank, T., "A Survey of Hardware Accelerators Used in Computer-Aided Design", in IEEE Design & Test, August 1984, pp 21-39

Butler, R. W., The Semi-Markov Unreliability Range Evaluator (SURE) Program, NASA TM 86261, Langley Research Center, Hampton, Virginia, July 1984

Castillo, Xavier and Siewiorek, Daniel P., "A Workload Dependent Software Reliability Prediction Model", in Digest of Papers of 12th Annual International Fault-Tolerant Computing Symposium (FTCS-12), June 1982

Cohn, R. B. et al., Ultrasystems, Inc., Definition and Trade-Off Study of Reconfigurable Airborne Digital Computer System Organizations, NASA Contractor Report 132552, Langley Research Center, Hampton, Virginia, 1974

Costes, A., Landrault, C., and Laprie, J. C., "Reliability and Availability Models for Maintained Systems Featuring Hardware Failures and Design Faults", in IEEE Transactions on Computers, Vol C-27, No 6, June 1978, pp 548-60

FAA FAR part 25, paragraph 25.1309(b), dated 5 August 1970

FAA Advisory Circular No. 25.1309-1, System Design Analysis, dated 7 Sep 82

Feller, W., An Introduction to Probability Theory and Its Application, Vol I, John Wiley & Sons, Inc., New York, 1966

Goldberg, H., Extending the Limits of Reliability Theory, John Wiley & Sons, New York, 1981

Hopkins, A. L., Smith, T. B., and Lala, J. H., "FTMP -- A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft", in Proceedings of the IEEE, Vol 66, No 10, October 1978, pp 1221-39

Iyer, R. V., and Rossetti, D. J., "A Statistical Load Dependent Model for CPU Errors at SLAC", in Digest of Papers of the 12 Annual International Fault-Tolerant Computing Symposium (FTCS-12), June 1982

Martin, D. J., "Dissimilar Software in High Integrity Applications in Flight Controls", in Software for Avionics, AGARD Conference Proceedings No 330, September 1982

McGough, J., and Swern, F., Bendix Corporation, Measurement of Fault Latency in a Digital Avionic Mini Processor, NASA Contractor Report 3462, Langley Research Center, Hampton, Virginia, October 1981

McGough, J., and Swern, F., Bendix Corporation, Measurement of Fault Latency in a Digital Avionic Mini Processor, NASA Contractor Report 3651, Langley Research Center, Hampton, Virginia, January 1983

McGough, J., Bendix Corporation, Feasibility Study for a Generalized Gate Logic Software Simulator, NASA Contractor Report 172159, Langley Research Center, Hampton, Virginia, July 1983

Menon, P. R. and Chappell, S. G., "Deductive Fault Simulation with Functional Blocks", in IEEE Transactions on Computers, Vol C-27, No 8, August 1978, pp 689-95

Meyer, John F., "Reliable Design of Software", in Rational Fault Analysis, Sacks and Liberty, eds., Marcel Dekker, Inc., 1977

Meyer, J. F., University of Michigan, Models and Techniques for Evaluating the Effectiveness of Aircraft Computing Systems, NASA Contractor Report 158993, Langley Research Center, Hampton, Virginia, July 1978

Nagel, P., Vought Corporation, Modeling of a Latent Fault Detector in a Digital System, NASA Contractor Report 145371, Langley Research Center, Hampton, Virginia, September 1978

NASA Contract NAS1-15428, Statement of Work for "Development and Evaluation of a Software Implemented Fault Tolerance (SIFT) Computer", Langley Research Center, Hampton, Virginia, July 1978

NASA Contract NAS1-17964, Task Assignment #5 -- Diagnostic Emulation Experiment, Langley Research Center, Hampton, Virginia, 20 May 1985

Ng, Ying-Wah, Reliability Modeling and Analysis for Fault-Tolerant Computers, University of California at Los Angeles, Engineering Document UCLA-ENG-7698, September 1976

Ratner, R. S., Shapiro, E. B., Zeidler, H. M., Wahlstrom, S. E., Clark, C. B., and Goldberg, J., SRI International, Design of a Fault Tolerant Airborn Digital Computer, Volume II - Computational Requirements and Technology, NASA Contractor Report 132253, Langley Research Center, Hampton, Virginia, October 1973

Smith, T. B., Hopkins, A. L., Taylor, W., Ausrotas, R. A., Lala, J. H., Hanley, L. D., and Martin, J. N., The Charles Stark Draper Laboratory, A Fault-Tolerant Multiprocessor Architecture for Aircraft, Volume I, NASA Contractor Report 3010, Langley Research Center, Hampton, Virginia, July 1978

Smith, T. B., "Fault Tolerant Processor Concepts and Operation", in Digest of Papers of the 14th International Conference on Fault-Tolerant Computing (FTCS-14), June 1984, pp 158-63

Stiffler, J. J., The Raytheon Corporation, Reliability Model Derivation of Fault-Tolerant, Dual, Spare Switching, Digital Computer System, NASA Contractor Report 132441, Langley Research Center, Hampton, Virginia, 1974

Szygenda, S. A. and Thompson, E. W., "Modeling and Digital Simulation for Design Verification and Diagnosis", in IEEE Transactions on Computers, Vol C-25, No 12, December 1976, pp 1242-53

Trivedi, K., Dugan, J. B., Geist, R., and Smotherman, M., "Hybrid Reliability Modeling of Fault-Tolerant Computer Systems", in Computing & Electrical Engineering, Vol 11, No 2/3, Pergamon Press, 1984, pp 87-108

VLSI DESIGN Staff, "1985 Survey of Logic Simulators", in VLSI DESIGN, March 1985, pp 74-80

Wensley, J. H., Lamport, L., Goldberg, J., Greem, M. W., Levitt, K. N., Melliar-Smith, P. M., Shostak, R. E., and Weinstodk, C. B., "SIFT: The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", in Proceedings of the IEEE, Vol 66, No 10, October 1978, pp 1240-54

Yount, L. J., "Architectural Solutions to Safety Problems of Digital Flight-Critical Systems for Commercial Transports", AIAA/IEEE Digital Avionics Systems Conference and Technical Display, Baltimore, Maryland, December 1984

-------- zones --- 5 --- 4 ------ 3 ------ 2 --

r \ n	3	4	5	6	7	8	9	10
5				.37	.51	.64	.75	.85
4			.45	.62	.76	.89	1.00	
3		.58	.78	.95	1.09	1.22		
2	.83	1.08	1.28	1.45	1.59	1.72		

Table 1 -- Ratio $\dfrac{\text{MTTF of r out of n system } (C_j = 1)}{\text{MTTF of constituent}}$

-------- all zones 9 or higher ------

r \ n	3	4	5	6	7	8	9	10
5				.35	.31	.27	.23	.21
4			.43.	.37	.31	.27		
3		.55	.45	.37	.31	.27		
2	.78	.60	.46	.38				

Table 2 -- Ratio $\dfrac{\text{MTTF of r out of n system } (C_1 = .9,\ C_{j\neq 1} = .1)}{\text{MTTF of constituent}}$

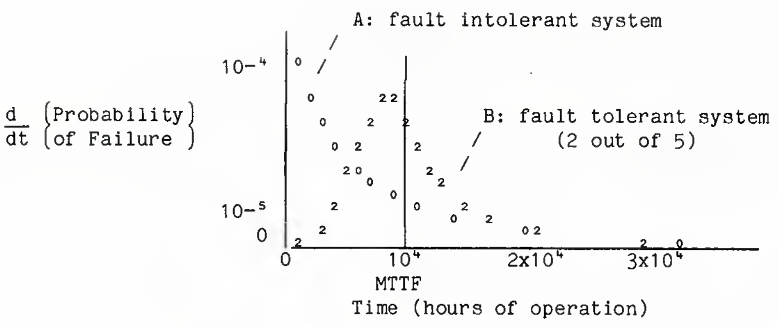


Figure 1 --- Failure Distributions With Same MTTF

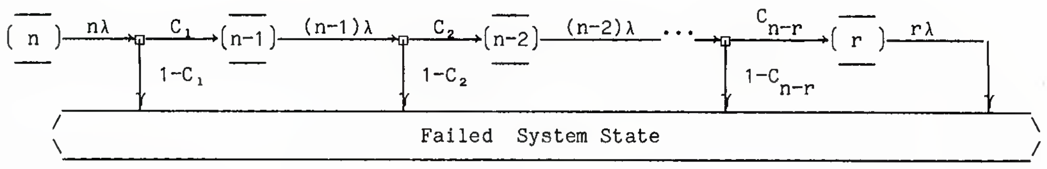


Figure 2 -- r out of n system with coverage parameters

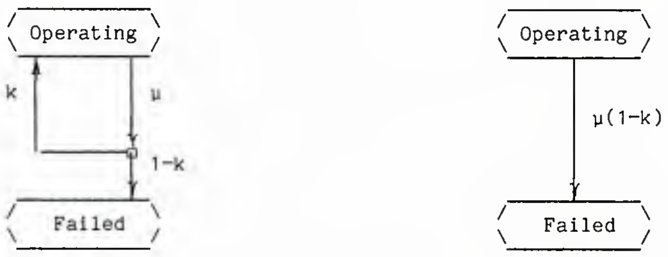


Figure 3a -- Fault Tolerant Software -- Figure 3b

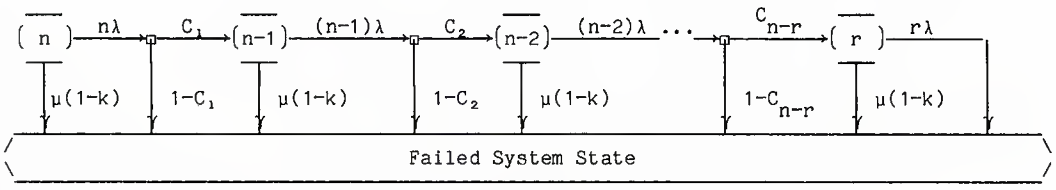


Figure 4 -- r out of n system with coverage parameters and a fault tolerant software scheme

SOME APPROACHES TO THE DESIGN OF HIGH INTEGRITY SOFTWARE

by

Professor J.T.Shepherd
Technical Director

Dr D.J.Martin

Mr R.B.Smith
Principal Engineer
Combat Aircraft Controls Division
GEC Avionics Ltd
Rochester, Kent ME1 2XX
UK

1.0 INTRODUCTION

As the complexity of aircraft systems has increased and the performance requirements for such aircraft have become more demanding the number of safety critical systems carried has increased.

This allied to the preponderance of digital systems on board the aircraft has meant that the software requirements of safety critical systems has become one of the pacing items in the development of new aircraft.

In the early days of high integrity systems analogue techniques were used and a variety of redundancy techniques were developed to cope with the need to obtain the required level of integrity from system elements whose inherent reliability was low.

With the advent of digital systems it is necessary to consider how such integrity can be achieved with the software of the system so that the total integrity of a safety critical system can be maintained.

There are in fact two methods which can be used to achieve this:-

a) fault avoidance
b) fault tolerance.

Of these, fault avoidance has been the main method used to date to achieve high integrity software and for example such digital flight control systems as are flying today have used fault avoidance as the main method of ensuring the integrity of the flight control software. Thus, the F8 experimental fly by wire system, the Jaguar fly by wire system, the YC14 AFCS and the L1011 ACS have all used fault avoidance techniques.

Fault avoidance techniques require the detailed application of structured design methods along with rigorous quality control and systematic testing of the software so that the probability of a software 'bug' being introduced or remaining undetected is extremely low. Such methods should of course be used in all software development and safety critical software is mainly distinguished from non safety critical software by the use of very small and very simple modules. Such modules are easy to verify and it is hoped that by such techniques that high integrity can be achieved.

In practice of course, no matter how carefully the software is designed it is impossible to establish that it is completely error free since the large number of possible states preclude exhaustive testing and the statistical analysis methods used in hardware development are not applicable to software.

Since a military aircraft is required to demonstrate a probability of failure due to say flight control failure of 1×10^{-7} per hour of flight and a civil aircraft requirement is two orders greater than this it is apparent that with fault avoidance techniques it is impossible to predict that the required level of integrity has been achieved. From this it follows that other methods which can be shown to be effective must be employed and thus fault tolerance methods must of necessity be considered.

In considering software fault tolerance methods, it is necessary to remember that the adoption of such methods should not mean that the design techniques should be abandoned. There is a wealth of data available to indicate that on average only 49% of the faults inherent in a program are discovered prior to entry of that software into service. While a good fault tolerance design might prevent the remaining faults from having a catastrophic effect on the system, the belated discovery of such faults and the resultant need to modify and reverify the program, increases the life cycle costs dramatically.

Thus fault tolerance methods and fault avoidance methods should go hand in hand. The resultant design should then be extremely reliable as well as giving the required level of integrity.

2.0 GENERAL CONSIDERATIONS OF SOFTWARE RELIABILITY

Failures in software stem from two major causes:

a) Design errors
b) Coding errors.

In practice of course, it is irrelevant to consider these categories as being different since both are due to errors made by the design and development team and a better classification is to consider faults in two categories:-

a) Faults occurring during the development and testing phases of the software programme.

b) Faults occurring during the inservice life of the software. This type of fault is known as a residual fault.

The prevention and discovery of the first category of faults is aided by the use of fault avoidance methods, notably:-

a) Top down structured design methods using such specification and design aids as SAFRA and MASCOT.

b) Structured programming.

c) The use of a block structured language such as CORAL 66.

d) The use of a powerful host computer during the development phase. The host used should have the ability to target the software to the final computer.

e) The use of a well developed and debugged compiler.

f) The use of structured walk throughs and good quality assurance techniques.

g) The preparation of good documentation.

h) The use of formal change control procedures.

These methods allied to extensive testing are the main techniques to be employed to provide fault avoidance and it is of interest that to date the organisations developing high integrity software, while in general following these practices have been reluctant to use high level languages.

The reason for this is that the key to fault avoidance is complete visibility and the lack of this in the compiler means that the development team cannot be assured that the compiler is error free. The use of fault tolerance techniques allied to the fault avoidance methodology should ensure that the number of residual faults are few and that the occurrence of these faults in service does not have a catastrophic effect on the system. When these techniques have been introduced, then high level languages can be considered.

3.0 THE FAULT AVOIDANCE APPROACH

As was stated earlier, fault avoidance techniques had been the main method used to date to provide high integrity software. To illustrate this, the following description of the software development for the Jaguar Fly by Wire system is used.

The essential requirement of high integrity software is visibility and this has been achieved by the use of simple software structures, clear requirements definition, thorough testing and design audit, detailed documentation and rigorous production and configuration control.

3.1 Flight Software Organisation for Jaguar Fly by Wire System

The real time control is achieved by a hardware Master Reset Timer which calls a non-interruptable Executive. The Executive then calls a number of Frames in a defined sequence designed to provide the required iteration rates for the various parameters. A Frame is a processing time slice containing related functional modules. One Frame typically contains signal selection, control law and logic module functions and consists of a set of program modules each of which defines a function that is easily defined, implemented, tested and audited.

The structure of the flight resident program is shown schematically in Figures 3-1

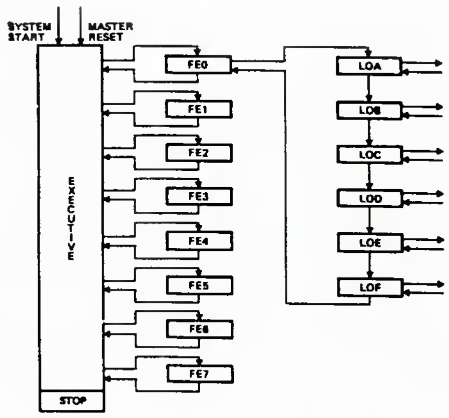


Figure 3-1 Control Flow - Executive & Frames

3.2 <u>Flight Software Development Process</u>

The flight software implements the aircraft control functions and meets the needs of the safety requirements. To ensure that both requirements are met, each stage of the design process was accompanied by formal testing and auditing.

Each design and test task was documented and design reviews were held at key stages of the software development to maintain a continuous check of integrity. Following initial design approval, configuration control was enforced to permit only authorised changes.

The key design requirements document for the Flight Resident Software (FRS) is the Software Requirements Document (SRD). The SRD controls the design implementation and has been the prime software interface between BAe and GEC Avionics Ltd.

The SRD was prepared in conjunction with BAe and it uses a mixture of English Language and program statements. These statements are intended to eliminate definition ambiguity. They form the basis of the definitive software design specification and are testable to prove the accuracy of the requirements definition.

The Software Structure Document (SSD) contains the running order of the modules within each program segment. The structure is designed to ensure that the chronological flow of data from input, through processing to output, is in strict sequence.

The codes of practice used in designing flight control software are contained in the Programmers Manual and the testing requirements for each phase are contained in the Testers Manual. These Manuals also define the procedures and documentation requirements for build standard, modification and quality assurance control.

The overall software development process is shown diagrammatically in Figure 3-2.

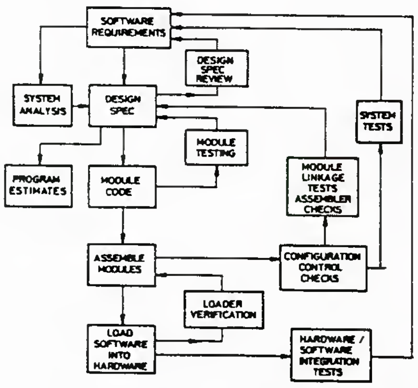


Figure 3-2 Software Development Process

4.0 METHOD OF INTEGRITY APPRAISAL

4.1 Introduction

The Integrity of the IFCS is primarily determined by the system architecture and therefore the primary elements of maximum concern are the points at which the redundant channels are consolidated or otherwise connected, together with the potential of a safety critical defect in the software or firmwhere.

A variety of methods of appraisal to assess flight worthiness were used, and these are as follows:-

i) General performance testing and safety assessment.

ii) Detailed performance testing and safety assessment of specific functions.

iii) Operational performance assessment.

iv) Design practices used for the specification and implementation of the function.

It was necessary to assess flight worthiness in a qualitative as well as a quantitative way, as many of the issues involved did not lead to useful quantitative estimates of safety critical risk.

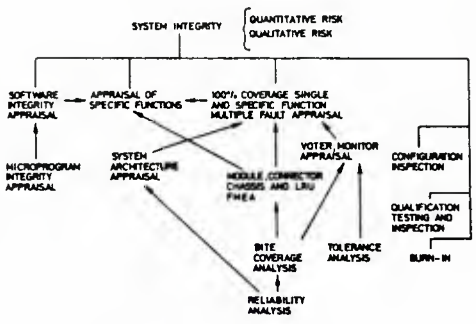

SYSTEM INTEGRITY
QUANTITATIVE RISK
QUALITATIVE RISK
SOFTWARE INTEGRITY APPRAISAL
APPRAISAL OF SPECIFIC FUNCTIONS
100% COVERAGE SINGLE AND SPECIFIC FUNCTION MULTIPLE FAULT APPRAISAL
MICROPROGRAM INTEGRITY APPRAISAL
SYSTEM ARCHITECTURE APPRAISAL
VOTER MONITOR APPRAISAL
CONFIGURATION INSPECTION
MODULE, CONNECTOR CHASSIS AND LRU FMEA
QUALIFICATION TESTING AND INSPECTION
BITE COVERAGE ANALYSIS
TOLERANCE ANALYSIS
BURN-IN
RELIABILITY ANALYSIS

Figure 4-1 Integrity Appraisal

The main elements of the appraisal/audit methodology, are shown in Figure 4-1 and summarised as follows:-

i) 100% coverage single fault FMEA.

ii) Multiple fault FMEA for specific combinations.

iii) Flight resident software integrity appraisal.

iv) Appraisal of specific functions.

v) Configuration inspection.

vi) Qualification programme.

vii) Burn-in programme.

The primary elements were supported by:-

a) Module, chassis and LRU FMEAs

b) Microprogram appraisals

c) Voter/monitor appraisals

d) Tolerance Analyses

e) BITE coverage analyses

f) System architecture analyses

g) Reliability analyses.

During the course of the integrity appraisal detailed technical evaluations of various features and functions of the IFCS were made. The requirements for these evaluations were generated mainly from the mainstream FMEA activity, and by BAe as a result of their engineering work. These evaluations were reported as a series of Technical Appraisals attached to the main integrity report and their results incorporated into the risk assessment.

The integrity appraisal was conducted by a team with specialist knowledge of the design of the equipment and to maintain the fidelity of the appraisal they reported to an independent authority which consisted of two senior engineers, one from GAv and the other from BAe.

5.0 RESULTS OF THE SOFTWARE APPRAISAL

The integrity appraisal had to deal with both hardware and software. Since this paper is concerned only with software, it is appropriate to consider only the software appraisal.

The objective of the software appraisal was to assess whether a software defect existed which, if activated, would lead to a safety critical event.

The software defects which may lead to a safety critical loss of control may arise from errors in the following:-

. Design Philosophy

. Requirements Definition

. Structure

. Frames

. Modules

. Hardware/Software Integration

. Support Software

. Configuration Production Control

The appraisal was designed to assess each error source in a variety of ways and the primary scope of each major activity is shown in Figure 5-1.

INTEGRITY ACTIVITY / ERROR SOURCE	A/C GROUND TRIALS	RIG AND EMULATION TRIALS	DIRECTED FLOW GRAPH	SYSTEM FMEA	FRAME TESTS	MODULE TESTS AND CHECKS	HARDWARE / SOFTWARE INT.	SUPPORT SOFTWARE FMEA	PRODUCTION / CONFIGURATION CONTROL FMEA	
DESIGN PHILOSOPHY	✓	✓								
REQUIREMENTS DEFINITION		✓		✓						
STRUCTURE	✓	✓	✓	✓	✓		✓			
FRAMES			✓	✓	✓					
MODULES			✓			✓				
HARDWARE/ SOFTWARE COMPATIBILITY	✓	✓		✓			✓			
SUPPORT SOFTWARE								✓		
CONFIGURATION CONTROL		✓		✓					✓	

Figure 5-1 Prime Method of Error Detection

The results of the GAv appraisals are given in turn.

5.1 Results of the system FMEA.

An extensive series of checks at system level were completed covering for example, the following:-

i) Executive and Synchronisation.

ii) BITE
- 1st line
- Pre-flight
- Failure identification
- Failure logic

iii) Control Functions
- Control Laws
- Signal flow
- Scheduling
- Transport delays
- Data management
- Voter/Monitors

iv) Workspace Usage

v) Storage and Run Time.

Further, a series of special case appraisals were completed, examples of which are as follows:-

i) One Frame not executed.

ii) Lane identify failures

iii) Sensor Tolerance analysis

iv) Asychronous operation.

The points which were considered to have a potential impact on system performance were assessed jointly by BAe and GAv. The result was that the deviations identified, either had no safety critical implications or were considered to present an insignificant safety critical risk.

5.2 <u>Results of the Frame Tests</u>

The Frame Tests checked for example:

i) That all the modules linked together in a Frame were defined by the SSD (Software Structure Document).

ii) The arithmetic accuracy of the Frame outputs.

iii) The operation of the failure logic was as defined in the SRD.

iv) The data module contained all the correct variables and constants.

The Frame Test deviations which had a potential impact on system performance were jointly assessed by BAe and GAv. None of the deviations had safety critical implications, or were considered to present a significant safety critical risk.

5.3 <u>Results of Module Tests and Technical Checks</u>.

The module tests checked for example that:-

i) The module code performs the function specified.

ii) The module is arithmetically correct.

iii) The module linkage is correct.

iv) The data module contained all the correct variables and constants.

These tests were complemented by a series of technical checks examples of which are as follows:-

i) The module requirement was checked for compatibility against the design specification.

ii) Conformity to the SRD, SSD and other manuals

iii) Module code is equivalent to specification language statements.

The thoroughness of these processes was assessed, and the results showed that a number of documentation errors and design deviations existed. However, no errors were found in the coded program and none of the deviations were considered to present a safety critical risk.

5.4. <u>Results of the Hardware-Software Compatibility Test</u>

An FCC 'Software ATP' was completed in order to test the compatibility of the FCC processor and I/O controller with the Flight Resident software. Although the tests checked for example, the control functions and the voter/monitors, no significant deviations were found.

5.5 Results of Support Software Appraisal

For the support software, a series of checks and tests were carried out to ensure the compatibility of the host processor software, with particular reference to the following programs:-

i) Macro Expander

ii) Assembler

iii) Simulator

iv) PROM Tape Copier

The results showed that the programs were mature and that all the deviations were due to the variations of the host machines used by the project and therefore the risk of a safety critical defect arising from the support software was considered to be insignificant.

5.6 Results of Software Production Procedures Appraisal

An appraisal of the software documentation and procedures was completed for the following:-

i) Production Methods

ii) Change Request Activity

iii) Module Documentation

The results of the appraisal showed that although a number of deviations were identified, the safety critical implications were considered to be insignificant.

5.7 Results of Configuration Control Appraisal

The configuration control appraisal checked for example that:-

i) Progress cards, Build Standards, Change Requests and other processes used in producing the various software issues had been completed.

ii) All queries have been answered and all necessary Change Requests had been entered on the Progress Cards and Build Standards.

This appraisal was carried out, for all Change Requests raised during the Jaguar FBW programme.

Although a number of documentation errors were identified, these were generally associated with documentation cross referencing and no errors have been identified in the coding.

5.8 Results of the BAe Queries Appraisal

The BAe Queries Procedure was established in order that points arising as a result of the Warton Rig Test Programme could be formally linked to the overall development programme for the software.

In order to check this procedure both GAv and BAe formally checked the response to each Query and no errors were identified which were considered to present a significant safety critical risk.

5.9 Conclusions from Software Appraisal

The software appraisal assessed its design, definition, implementation, test support and configuration control but the joint survey did not identify any defect which would result in a safety critical condition.

However, in order to give an indication of the sources and types of software defect identified during the project, an analysis of the module change requests was completed.

These have been broken down into the following categories:-

Design	44%
FMEA	24%
Rig errors	12%
Code errors	7%
Not required	13%

This shows that the most common reason for a change request was modification to the design requirements specification.

Further, the total distribution of all CRs clearly indicated that the overall level of activity declined after June 1979 and that the software was mature at first flight.

The appraisal also highlighted the following:-

. <u>Design</u>

  The number and distribution of the CRs, raised as the result of design or similar errors, followed a predictable pattern, in that the peaks corresponded to each major issue of the assembled program.

. <u>Code</u>

  The small percentage of coding errors indicate the thoroughness of the module and program testing carried out during and after module assembly. It is also significant that the majority of the errors were detected before the first development assembly. Only three coding errors were found after the first assembly had been issued and the program has been free of code errors since June 1980.

. <u>Module FMEA</u>

  The number of CRs raised as the result of the module technical check FMEA indicates that the procedure is a significant method of error detection.

. <u>Rig Test Results</u>

  The testing of the modules as part of the assembly was carried out on the Warton Rig and the resultant CRs from this work indicate the necessity of such equipment. The significant volume of error detection occurred after the first assembly was tested on the Rig.

However, it was not possible to draw any significant quantitative evidence that the defect rate of the software was compatible with a safety critical risk of 1 x 10^{-7} per flight hour.

The final judgment on the software was therefore, that it had been audited by a variety of different methods encompassing its performance, construction and code and no safety critical defect had been identified. Therefore, it was concluded that the software was sufficiently safe to permit flight trials to commence.

6.0 METHODS OF PROVIDING FAULT TOLERANT SOFTWARE

6.1 Basic Approach to Fault Tolerant Software

Any fault tolerant system be it hardware or software must contain the following elements:-

a) A method of detecting a failure.

b) A method to determine which channel of the system has in fact failed.

c) An alternative channel which can take over the operation of the system until such time as the failed channel can be repaired.

In hardware, the usual approach to fault tolerance is to provide a number of redundant channels. In this case, errors are detected by a comparison between the outputs of the 'N' identical channels and any channel which differs from the average of the outputs is adjudged to be the failed channel. This channel is then isolated and the remaining channels continue to operate at a lower level of integrity.

The major problem with software is that this basic method is unsuitable. This is because the basic concept of hardware redundancy relies upon spatially separating the channels so that the probability of a failure in one channel causing further failures in the other channels is extremely low. Since a software fault is essentially a latent fault it exists in all identical pieces of software and will be triggered by a combination of input data and the internal state of the program.

Thus, if all channels of a high integrity system contain identical software, then if the systems are synchronised (as is necessary for the hardware monitoring technique to work), then all of the channels will experience a simultaneous software failure.

From this it follows that an alternative method must be followed to achieve fault tolerant software.

The most obvious method of providing fault tolerance is then to provide alternative versions of the program. This method requires a different version of the program for each channel and since a high integrity system requires at least three channels there must be at least three versions of the program produced.

The difficulty of a single programming team providing these alternative versions of a program has led to the concept of independent programming teams. This means that for the simple triplex case three independent programming teams are required.

An alternative approach is the error recovery block technique first proposed by Horning et al in 1974. ("A Program Structure for Error Detection and Recovery". Lecture Notes in Computer Science 16, Ed. E Gelenbe and C Kaiser, Springer-Verlag, Berlin, 1974). This method establishes a series of recovery points inside the program. Upon detection of a failure, the program automatically returns to the nearest recovery point and executes an alternative version from that point.

To detect a failure an acceptance test is associated with each recovery block and the algorithm for the error recovery block scheme is:-

```
ENSURE         Acceptance Test
BY             Primary Module
ELSEBY         Alternate Module 1
ELSEBY         Alternate Module 2
   .
   .
   .
   .
ELSEBY         Alternate Module n
ELSE error
```

It will be seen that the recovery block method is a more formal method of 'N' version programming applied to modules of a program rather than to the whole program. As such it still requires a number of independent programming teams and the problems of establishing and controlling such teams remain.

One of the characteristics of the error recovery block method is that the various alternates provide standby redundancy and are only invoked in the event of the primary module failing the acceptance test. This minimises the execution time during the normal operation of the program but a failure may cause the program to stop operation until the system has recovered. While this is acceptable for many systems, for such systems as flight control it is necessary for the system to continue operation.

6.2 The Dissimilar Software Approach

As was discussed above, fault tolerance requires the provision of dissimilar software. This approach was adopted for the A310 Slats and Flaps system and this is discussed in more detail below.

Secondary flight control systems such as spoilers/airbrakes, automatic trim, slats and flaps and even some primary functions such as yaw damping, have throughput requirements which are much less onerous than primary flight control. Computer units for these applications are capable of being implemented using any of a number of 8 and 16 bit microprocessors.

These systems, though relatively undemanding in computing power, nevertheless have high integrity requirements.

Failure survival constraints are imposed upon the electronic control system since there are no mechanical links from the pilot's controls to the surfaces, i.e. it is a fly-by-wire system. The safety constraints are defined by probabilities for various occurrences, including:

1) Inadvertent deployment of the slats or flaps must have a probability of less than 10^{-9} per flight hour.

2) Slats or flaps no longer operating and no warning given to the pilot, must have a probability of less than 10^{-9} per flight hour.

3) Slats not operable must have a probability less than 10^{-5} per flight hour.

These constraints relate to the entire slat and flap operating systems and include the electronic, mechanical and hydraulic components. Hence the requirement for safe operation of the SFCC has to be better than the above figures.

In considering candidate system architectures to achieve the stated requirements, one of the major considerations was the comparative simplicity of the task in relation to other flight control tasks. Operation of the slats and flaps as performed by the software, is mainly a sequence of logical expressions rather than arithmetic expressions and complex filters as would be found in typical autopilots and autostabilisers. This results in throughput and instruction set requirements that can be readily achieved by commercially available microprocessors. The use of microprocessors allows a significant cost saving when compared to such a system using a purpose build processor. However, the disadvantage of microprocessors in high integrity applications is that their internal workings are not visible. Thus the failure mechanisms of such processors cannot be predicted.

Another major factor in the choice of an architecture is the views of the certification authorities. Lack of experience in the industry and the difficulty in assessing the integrity of software has led to recommendations from the authorities that consideration be given to:

a) the use of monitoring, limiting or other provisions which are independent of the digital computation, to reduce the effect of failure within it.

b) the use of dissimilar elements in critical portions of the equipments, particularly where analysis may be difficult or inconclusive (e.g. the processor).

The third major factor is the failure survival requirements. In a control system such as on the Jaguar fly-by-wire aircraft, it is not enough just to identify that there has been a failure. It is also necessary to continue to work correctly after the failure. Hence the necessity to identify where the failure is and to isolate or absorb it. However, for the A310 Slats and Flap System it is sufficient to know that there has been a failure. On recognition of the failure, there are brakes in the wings which are operated to freeze the surfaces in their current position. Although this may mean a slatless or flapless landing, or indeed, an immediate return to the airport from which the aircraft has just been taking off, it does not prevent the continued safe flight and landing of the aircraft. This necessity for fail-safe capability is reflected in the safety requirements as outlined above, item 3 (slats not operable: probability $<10^{-5}$ per hour) being more an availability than a safety constraint.

It was concluded, therefore, that two different microprocessors should be used in parallel to perform the slat and flat operating task. This makes the probability of a common internal failure or design error within the microprocessors extremely unlikely. It also ensures dissimilarity of the software at the code level.

Although this was felt to increase the level of confidence that a common coding error would be minimal, it does not resolve the problem of common software structure or algorithmic design errors. Hence it was decided to perform two complete software development tasks, one for each microprocessor, with only the system requirements as produced by the customer being common to the two.

6.2.1. System Description

A schematic diagram of the computer architecture is shown in Figure 4.6. Two computers are required in order to meet the availability requirements.

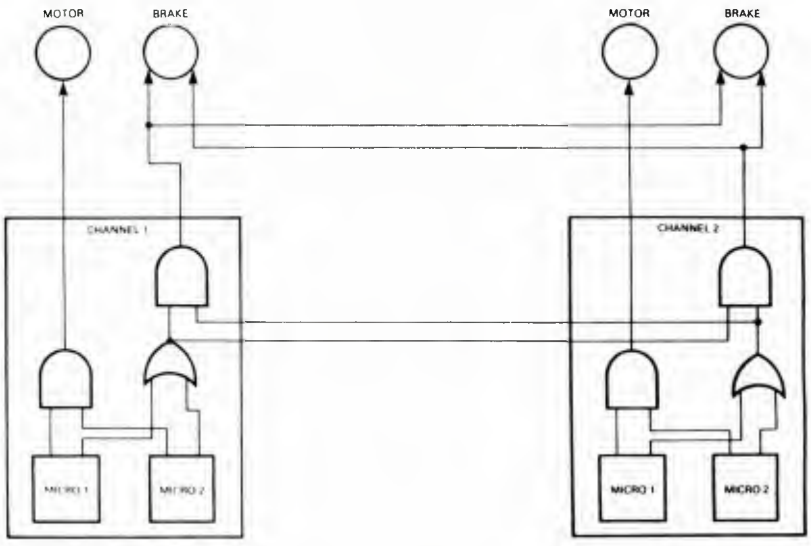


Figure 6.1 Computer Architecture Schematic

Internally, each of the computers contains two different microprocessors. A computer will only drive its hydraulic motor if both microprocessors agree, otherwise the motor output is locked leaving the other computer to drive the system.

If a computer fails, this will be detected by the different outputs from microprocessors 1 and 2 and the computer will then isolate itself.

If there is an uncommanded movement of the output e.g. a torque shaft breaks, then both computers will detect the incorrect deployment and will operate the wing brakes. This freezes the flaps/slats in their current position.

6.2.2 Software Development for Dissimilar Redundancy

A separate Software Requirements Document (SRD) is produced for each lane. Having produced the SRD, the software development procedure then follows the normal path, in each lane, of top-down analysis to produce a modular structure and to design and generate the code for each module.

At this stage, instead of embarking on module testing, the approach that has been taken is to assemble the software for each lane and then to perform hardware/software integration testing. It is felt that, since the lanes must agree to provide the computing function they each provide for the other the most stringent test environment. This test philosophy is amplified in section 6.2.4.

To avoid the possibility of design errors being introduced by a common assembly fault, two different host computer facilities are used to assemble code for the two processors.

The choice of microprocessors from two different suppliers further reduces the risk of the associated assembler packages having common errors.

Each microprocessor memory is loaded from the associated development system in a dedicated format. These formats, which are different, are read by the PROM programmer which is independently checked for correct function.

The two software development activities are kept completely separate, the two programs eventually being proved by integration with the hardware.

A brief summary of the software development process is shown in Figure 6.2.

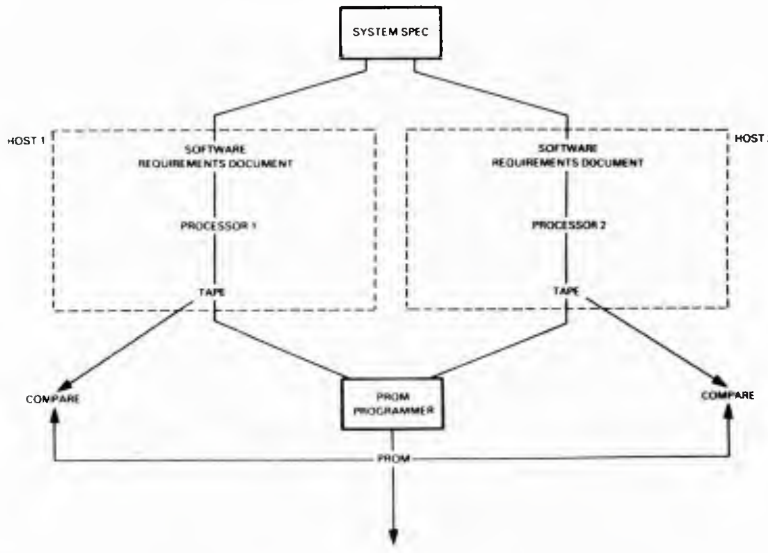


Figure 6.2 Dissimilar Software Development

6.2.3. <u>Design Methodology</u>

One disadvantage of trying to verify software by testing is that it is extremely difficult to prove that the software is free from design errors. In a similar redundant system the probability of an undetected software design error adversely affecting the system safety must be compatible with the integrity objective.

From the Airbus Industrie A310 SFCC, the software has been prepared twice, using two different software development facilities. The flight resident software suites thus produced are executed by different, asynchronously operated microprocessors. The outputs of the two microprocessors are continuously compared and any difference greater than a defined threshold causes the system to disconnect after a preset time delay with all output drives being removed.

A design error in one of the two dissimilar lanes can never produce a hazardous output from the system.

Design errors in both dissimilar lanes could only produce a hazardous output if their resultant effect was identical and occurred within the pre-set time delay of the cross-lane comparison.

The benefits of dissimilarity are clear. The precautions that have been taken to ensure that dissimilarity is maintained are listed below.

. Software for each of the two processors is prepared by two different groups.

. The processors are different for the two lanes - are made by different manufacturers and have different instruction sets.

- The two suites of software are prepared and assembled using two different software facilities - also made by different manufacturers.
- The processors obey instructions which have different object codes and are located in differently mapped program stores.
- The data used by the processors is located in data stores which are also differently mapped.
- The processors have separate clocks and operate asynchronously so that there is no requirement for frame synchronisation to link them together.
- The program store for each processor is loaded using a PROM tape, the format of which is different for the two lanes, since each is generated by a different support system.
- Discrete words exchanged cross-lane for monitoring have their bit pattern deliberately shifted. This avoids the possibility of a common error when comparing the two lane outputs in each lane.

With these precautions it is unlikely that the same software design error can occur in both lanes to produce the same hazardous and undetected output simultaneously from both lanes.

6.2.4 Software Verification

In a similar redundant system where software is common to all lanes, a single software error could cause loss of the total system. A software verification procedure has therefore been developed which reduces the software risk to acceptable proportions.

Several of the techniques thus developed incur no significant cost penalty and can be applied directly to the dissimilar approach.

With other techniques such as module testing, the extent to which they are applied depends on the complexity of the module function and the consequences of not detecting errors. These techniques have to be critically reviewed and applied as necessary to achieve the required level of integrity.

Testing
The duplication of software is considered to provide sufficient integrity to meet the safety requirements of a system in which availability is not a predominant requirement.

Nevertheless, it was felt that to provide the required availability, certain module tests needed to be performed. The modules are divided into three categories:-

a) critical modules which will be subjected to full testing e.g. those that drive system outputs or monitor for asymmetrical deployment or runaways.

b) modules that will require supplementary testing because the software/hardware integration tests do not accurately check certain thresholds.

c) modules requiring no testing other than that performed during hardware/software integration.

6.3 An Alternative Approach to Fault Tolerant Software

It has been seen that fault tolerant software requires the development of alternative versions of the flight program. The A310 system follows this pattern and it has been seen that two independent software teams are needed.

An alternative to this approach was proposed by Shepherd while at the Cranfield Institute of Technology.

This approach is currently being examined by C.I.T. and GEC Avionics under contract from the Royal Aircraft Establishment.

Essentially, the method follows the general concept of error recovery blocks but evaluates the alternate block concurrently with the primary block.

The method depends upon two techniques:

a) The concept of Temporal Separation of Software Channels.

b) The concept of deriving an alternate version of the program from the primary version.

These techniques are discussed below.

6.3.1 Temporal Separation

The concept of temporal separation is essentially based upon the fact that while spatial separation prevents the proliferation of hardware faults, software faults are time dependent. Thus, to ensure that a software fault does not affect all channels simultaneously, it is necessary to ensure that each channel does not process identical data. The method of ensuring this depends upon the fact that for a real time system such as flight control a high iteration rate is used. The method is then to use inputs from different time periods for each of the channels.

To illustrate this consider a triplex system. Assume that each channel has program version A as its software and that the total system has reached iteration n.

Then let An represent program A on its nth iteration, An-1 program A on its n-ith iteration etcetera.

Then let the distribution of programs be as follows:-

Channel No.	Program State
1	An
2	An-1
3	An-2

The system operates on channels 2 and 3. If An can be checked for correctness then on the next iteration this becomes An-1 and since it has been established that the program operates correctly with this input data (on the previous iteration), it is safe to control the system using this software and input data.

To decide whether An is correct it is necessary to provide a method of checking An. This is done by generating an alternate version of the program using the methods described below.

For the moment however, assume that an alternate program B exists. Then using the same notation as given above the basic system implementation is given by:-

Channel No.	Primary Program	Alternate
1	An	Bn
2	An-1	Bn-1
3	An-2	Bn-2

If then An is compared with Bn and is within tolerance then the system is operating correctly and the data can be processed to control the system using channels 2 and 3.

If An and Bn outputs disagree by more than a set tolerance then a Lagrangian Extrapolation of previous outputs can be made.

Thus from An-3, An-2 and An-1 a value An(est) is obtained. Similarly, a value Bn(est) is found.

The following comparisons are then made:-

$$\begin{aligned} An &- An(est) = E1 \\ An &- Bn(est) = E2 \\ Bn &- An(est) = E3 \\ Bn &- Bn(est) = E4 \end{aligned}$$

If then $(E1 + E2)/2 < (E3 - E4)/2$ then program A is correct else program B is correct.

In practice an additional check is carried out to ensure that the error values of the smaller of the above equations are below a tolerance level.

This method is valid for such systems as flight control since as was mentioned above a high iteration rate is used and the difference between successive outputs is small. Thus, it is possible to extrapolate to obtain the estimates required.

In practice, the following implementation was carried out:-

Channel No.	Main Program	Alternate
1	An	Bn-2
2	An-1	Bn
3	An-2	Bn-1

and a state table established showing previous states.

Then if the new value of Bn-2 is different from the previous value of Bn-1 a hardware fault must exist in channel 1. Similar considerations apply to channels 2 and 3. Thus the method can be used to detect both hardware and software failures.

For this method to function it is of course necessary to construct an alternate version of the program. The method proposed to do this is described below.

6.3.2. A Method of Generating an Alternate Version of a Program

In considering software monitoring, it is necessary to consider the four types of operation encountered in a program. These are:-

a) Arithmetic operations
b) Logical operations
c) Decision and branch operations
d) Input/output operations (including interrupts).

Each of these types of operation are considered in turn.

6.3.2.1 Arithmetic operations range from the simple assignment operation to very complex functions. It is obviously necessary to consider this total range of operations.

Assume therefore, that the program contains a series of arithmetic modules and it is required to check these modules.

Let the output from a program module A be a function of variables B1, B2, ..., Bm.

Then at sample n assume

$A_n = f(B_{1n}, B_{2n} \dots, B_{mn})$ is correct.

At sample n+1

$A_{n+1} = f(B_{1n+1}, \dots, B_{mn+1})$

A_{n+1} differs from A_n by a value ΔA_{n+1} and this change is due to the change in the values of B_{1n} to B_{1n+1} of ΔB_{1n+1}.

Then

$$A_n + \Delta A_{n+1} = f((B_{1n} + \Delta B_{1n+1}), (B_{2n} + \Delta B_{2n+1}) \dots (B_{mn} + \Delta B_{mn+1}))$$

If now a relationship can be found such that

$$f((B_{1n}, \Delta B_{1n+1}), \dots (B_{mn} + \Delta B_{mn+1})) = f(B_{1n}, \Delta B_{2n} \dots B_{mn}) + g$$
$$(B_{1n}, \Delta B_{1n+1}, B_{2n}, \Delta B_{2n+1} \dots B_{mn}, \Delta B_{mn+1})$$

Then

$$\Delta A_{n+1} = g(B_{1n}, \Delta B_{1n+1} \dots, B_{mn}, \Delta B_{mn+1})$$

But $\Delta A_{n+1} = A_{n+1} - A_n$

Thus, if the function g is obtainable, two measures of A_{n+1} are available and if An is correct then the correctness of A_{n+1} can be established.

The problem must therefore be considered in two parts:-

a) For a given function f, does a derived function g obeying the above requirements exist?

b) If such a function g exists, is it independent of f, (i.e. if f fails does g survive)?

It is obvious that for many of the commonly used functions in realtime systems, it is possible to derive such a g function and to show that they are independent.

6.3.2.2 <u>Logical Operations</u>

In addition to arithmetic operations it is necesary to consider logical operations of the form:

$A := B + C$

$A := B.C$

where $+$ = OR

$.$ = AND

In general with these types of equations, it is easy to produce an alternative algorithm.

De Morgans' theoram states that

$$\overline{A.B} = \overline{A} + \overline{B}$$
$$\overline{A + B} = \overline{A}.\overline{B}$$

Thus the above functions can be expressed as:-

$$A = B + C = \overline{\overline{B + C}} = \overline{\overline{A}.\overline{B}}$$
$$A = B.C = \overline{\overline{B.C}} = \overline{\overline{B} + \overline{C}}$$

In general, as is common in logic design any Boolean function can be expressed in terms of either NOR or NAND operations and thus there are always two alternative expressions for logical operations. Thus the requirements for having different algorithms for the main and check programmes are satisfied.

6.3.2.3 Decision and Branch Operations

These operations are typical of the form:

```
BEGIN
    If X = Y  THEN BEGIN
              MODULE A
              END

              ELSE BEGIN
              MODULE B
              END
END
```

It will be seen that this type of operation is a combination of arithmetic and logical operations and can be treated as such.

Thus, the next iteration of the loop is

```
If  X + ΔX = Y + ΔY  THEN ......
```

This is modified to

```
BEGIN
IF    X - Y < > ΔY - ΔX  THEN BEGIN
                              MODULE B
                              END
                         ELSE BEGIN
                              MODULE A
                         END
END.
```

It will be seen that the previous values are compared with the increments and the opposite state (not equal instead of equal) to give the branch condition. Once again therefore, different algorithms are used to check the main algorithm.

6.3.3 Results of Preliminary Study

In order to check the validity of the approach, software representing the short period mode of an aircraft was developed and the techniques described above were applied. Approximately 60,000 random faults were introduced for each of 3 different input signals (which had random noise superimposed to ensure that realistic signals were used).

The results obtained were:-

	Sine Test	Triangle Test	Sawtooth Test
No. of faults	61127	62019	61063
% Detected	100	100	100
% Errors corrected	83.9	82	76.5
Acceptable errors (1-15%)	16.0	17.9	24.2
Unacceptable errors	0.1	0.1	0.1

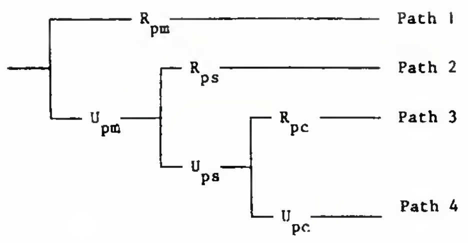
In practice the sinusoidal input is more nearly representative of the type of input used in flight control.

It should be realised that no attempt was made to optimise iteration rates or tolerance levels during this study and that therefore it is to be expected that even better results can be obtained.

The results obtained to date however indicate that the proposed approach is valid and that it is worth considering the concept further.

6.3.4 Estimates of Failure Probability

We now pass on to considering the probability of failure of the fault tolerant system. A success/failure disagram of the fault tolerant software can be constructed as follows: -



where R_{pm} = the reliability of the main program

U_{pm} = the unreliability of the main program

R_{ps} = the reliability of the difference equation program

U_{ps} = the unreliability of the difference equation program

R_{pc} = the reliability of the check program used in the fault tolerance system

U_{pc} = the unreliability of the check program.

Each of the paths shown in the diagram describes a particular situation in the fault tolerant software.

Path 1 reliability = R_{pm}

Path 2 reliability = $U_{pm}*R_{ps}$

Path 3 reliability = $U_{pm}*U_{PS}*R_{pc}$

Path 4 = $U_{pm}*U_{ps}*U_{pc}$

Path 4 will result in a failure in the fault tolerant software.

The probability of a software failure of a program is closely linked to the unreliability of a program. We can say that

$P_{sf} = P_{sfmain} * P_{sfsecondary} * P_{undetected\ error}$

For a typical piece of software

$P_{sf} = 1 \times 10^{-3}$/hour

for well tested programs.

Assuming that the main and secondary programs of the system are well tested, we can say that

$P_{sf} = 10^{-3} * 10^{-3} * P$ undetected error

From the results obtained it is possible to assume that the probability of an undetected error will be small. A rough estimate could be

$P_{undetected\ error} = 10^{-3}$/hour

and the probability of a software failure for the fault tolerant system suggested becomes

$P_{sf} = 10^{-9}$/hour

despite each section of the software having a probability of failure

= 10^{-3}/hour.

Acknowledgement
Part of this work has been carried out with the support of the Procurement Executive, Ministry of Defence.

ROBUST CONTROL SYSTEM DESIGN

by

J. Ackermann
Director of DFVLR Institute for Flight System Dynamics
DFVLR Oberpfaffenhofen
8031 Wessling
W.-Germany

SUMMARY

The short period longitudinal mode of an F 4-E with horizontal canards is unstable in subsonic flight and unsufficiently damped at supersonic speed. The control system has to provide acceptable pole locations according to military specifications for flying qualities. A fixed gain controller using three paralleled gyros is designed, such that the pole region requirements in four typical flight conditions are robust with respect to gain reduction to one third. Thus nothing bad happens immediately after one or two gyro failures. Failure detection and redundancy management may be performed at a higher hierarchical level, which does not have to be extremely fast. The use of accelerometers or air data sensors for angle of attack or dynamic pressure is totally avoided in this concept and no gain scheduling is necessary. The design for robustness with respect to different flight conditions and sensor failures is performed by a novel parameter space design tool.

I. INTRODUCTION

Redundancy management in control systems is usually viewed separately from the control algorithm. The control system is designed under the assumption, that sensors do not fail. Then redundancy management has to provide the required measurements with only very short interruptions by failures of individual sensors. If the plant is for example an unstable aircraft, this means that failure detection is vital for stabilization, it has to operate fast and this requirement is in conflict with the requirement of low probability of false alarms.

In this paper a hierarchical system is proposed. Its basic level is a fixed gain control system, which is designed such, that pole region requirements are robust with respect to component failures. All more sophisticated tasks like failure detection and redundancy management, plant parameter identification and controller parameter adaptation or gain scheduling are assigned to higher levels, if they are required for best performance. The higher levels processes more information and are operating in a slower time scale than the basic level. Since the higher levels are not vital for stabilization they can make their decisions without panic haste.

This paper deals with the design of the robust basic level control system. The particular example is an F 4-E, which is destabilized by horizontal canards, see Fig. 1. Only the short period longitudinal mode is considered, i.e. second order dynamics. The actuator is modelled as a first order low pass with transfer function 14/(s + 14), its state variable is δ_e, the deviation of the elevator deflection from its trim position. δ_e is not fed back, because this would require an estimate of the trim position.

In a previous study [1], [2] measurement of normal acceleration N_z and pitch rate q is assumed and the linearized state equations are written in sensor coordinates with the state vector $\underline{x}^T = [N_z \quad q \quad \delta_e]$. Thus

$$\dot{\underline{x}} = \underline{A}\,\underline{x} + \underline{b}\,u \tag{1}$$

$$\underline{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 14 \end{bmatrix} \qquad \underline{b} = \begin{bmatrix} b_1 \\ 0 \\ 14 \end{bmatrix}$$

Data for the four typical flight conditions of Fig. 2 were taken from [3] and are given in the appendix. The eigenvalue locations of the short period mode are given in table 1.

Table 1

FC	MACH	ALTITUDE	OPEN LOOP SHORT PERIOD EIGENVALUES	
1	0.5	5000'	-3.07	1.23
2	0.85	5000'	-4.90	1.78
3	0.9	35000'	-1.87	0.56
4	1.5	35000'	$-.87 \pm j4.3$	

The aircraft is unstable in subsonic flight and unsufficiently damped in supersonic flight, such that adequate handling properties must be provided by the control system.

Note that in stationary flight the elevator and canard are not used independently. The commanded deflections are coupled as

$$\delta_{ecom} = u$$

$$\delta_{ccom} = -0.7u$$

where the factor -0.7 was chosen for minimum drag. Thus the short period mode stabilization is a single-input problem.

The required closed loop eigenvalue locations are given by military specifications for flying qualities of piloted airplanes [4]. For the short period mode described by

$$s^2 + 2\zeta_{sp}\omega_{sp}s + \omega_{sp}^2 = 0 \tag{2}$$

the restricted range of damping ζ_{sp} and natural frequency ω_{sp} is

$$0.35 \leq \zeta_{sp} \leq 1.3 \qquad \omega_a \leq \omega_{sp} \leq \omega_b \tag{3}$$

where ω_a and ω_b depend on the flight condition and are given in the appendix for the four conditions considered here.

Fig. 3 shows the nominal region Γ_j, eq.(3) together with the open loop eigenvalues for a subsonic flight condition j. Damping greater than one in eq.(3) corresponds to two real eigenvalues. Eq.(3) would admit some real pairs of poles with one of them outside the region Γ_j. In the following no use is made of this possibility. For all real pairs inside Γ_j condition (3) is satisfied. We require, that the closed loop short period poles of each flight condition j = 1, 2, 3, 4 are located in the respective region Γ_j.

The military specifications do not contain requirements for the location of additional closed loop poles originating from actuator or feedback dynamics. Quick response is essential for a fighter,therefore the non short period eigenvalues should not unnecessarily slow the dynamic response. In order to keep them fast enough and separate from short period eigenvalues an additional region to the left of Γ_j is prescribed. The damping requirement $\zeta \geq 0.35$ is kept from eq.(3) and a natural frequency range $\omega_b \leq \omega \leq \omega_d$, $\omega_d = 70$ rad/sec is chosen in order to maintain a bandwidth limitation below the first structural mode frequency. The extended region is shown in Fig. 3.

The assumed type of sensor failure is that the nominal gain v = 1 is reduced to some value $0 \leq v < 1$ and an additional bias or noise term is added at the sensor output. As far as eigenvalue location is concerned, only the gain reduction to zero is important.

The objective of this paper is to design the basic level control system such that the pole region requirements of Fig. 3 are robust with respect to changing flight conditions and sensor failures. A novel "$\mathcal{K}$-space" design technique [5] is applied in this design. It will be reviewed briefly in the following paragraph. In application to the example it is then shown, how robustness with respect to changing flight conditions can be achieved by appropriate choice of k_{Nz} and k_q in an output feedback control law

$$u = -[k_{Nz} \quad k_q \quad 0]\,\underline{x} \tag{4}$$

For robustness with respect to sensor failures in [1], [2] a configuration with two gyros and one accelerometer and dynamic feedback was studied. It showed the disadvantage of using the accelerometer. Therefore here a different solution with three gyros and dynamic feedback is given. For this solution the responses in C* for a pilot step input are given, where

$$C^* = (N_z + 12.43q)/C_\infty \tag{5}$$

The stationary value C_∞ is used for normalization.

II POLE REGION ASSIGNMENT

The most essential aspect of $\mathcal{K}$-space design is pole region assignment. Other features will be discussed later using the design example. If a tradeoff with other design requirements has to be made it is not satisfactory to find <u>one</u> solution, for which all eigenvalues are in their respective regions in s-plane, e.g. by pole placement or root locus techniques. It is desirable to find <u>all</u> such solutions. This is achieved by mapping the region Γ in s-plane into a region P_Γ in the parameter space $\mathcal{P}$ of coefficients of the desired characteristic polynomial first. Then P_Γ is mapped into a corresponding region K_Γ in the parameter space of feedback gains. The first step only deals with properties of polynomials

$$\begin{aligned} P(s) &= p_o + p_1 s + \ldots + p_{n-1} s^{n-1} + s^n \\ &= [\underline{p}^T \quad 1]\,[1 \quad s \ldots s^n]^T \qquad (6) \\ &= \prod_{i=1}^{n} (s - s_i) \end{aligned}$$

The problem is: Find the region P_Γ in $\mathcal{P}$ space such that $\underline{p}^T \in P_\Gamma$ if and only if $s_i \in \Gamma$ for $i = 1, 2 \ldots n$. The boundaries of P_Γ for a connected region Γ with two real axis intersections at σ_L and σ_R consists of three parts corresponding to the cases that a real eigenvalue crosses the boundary in s-plane at σ_L or at σ_R or a complex conjugate pair crosses the complex boundary. For the real values these boundaries in $\mathcal{P}$ space are the n-1 dimensional hyperplanes $P(\sigma_L) = 0$ and $P(\sigma_R) = 0$.
For the complex case

$$\begin{aligned} P(s) &= (s - \sigma - j\omega).(s - \sigma + j\omega) \cdot R(s) \\ &= [s^2 - 2\sigma s + \sigma^2 + \omega^2(\sigma)] \cdot R(s) \end{aligned} \qquad (7)$$

If for example the boundary $\omega^2(\sigma)$ is a circle with real center σ_o and radius r, i.e. $(\sigma - \sigma_o)^2 + \omega^2 = r^2$, then $P(s) = [s^2 - 2\sigma s + r^2 + 2\sigma_o\sigma - \sigma_o^2]\, R(s)$. For a fixed $R(s)$, $\underline{p}^T$ depends linearly on σ, i.e. in $\mathcal{P}$ space a straight line segment from a point on $P(\sigma_L) = 0$ (with a double root at σ_L) to a point on $P(\sigma_R) = 0$ (with a double root at σ_R) results. As the n-2 coefficients of R(s) vary, this straight line is moved and forms the complex boundary in $\mathcal{P}$-space. For n = 3 this is illustrated by Fig. 4. If $\underline{p}$ crosses the plane containing the triangle ABC, then a real root crosses the unit circle at s = -1 and analogously for the triangle BCD and s = 1. If $\underline{p}$ crosses the hyperbolic paraboloid, which is formed by a family of straight lines, then a complex conjugate pair of eigenvalue crosses the unit circle. If $\omega^2(\sigma)$ is a conic section

$$\omega^2(\sigma) = c_o + c_1\sigma + c_2\sigma^2 \qquad (8)$$

then the image for a fixed R(s) is a conic section instead of the straight line above. The complex boundary may be defined piecewise as in Fig. 3.

The complex and the two real boundaries partition the $\mathcal{P}$ space into regions distinguished by the location of the eigenvalues relative to Γ.

In the second step a controller structure is assumed, e.g. state feedback

$$u = -\underline{k}^T \underline{x}\,, \quad \underline{k}^T = [k_1 \quad k_2 \ldots k_n] \qquad (9)$$

and the region P_Γ is mapped into a region K_Γ in the controller parameter space $\mathcal{K}$ with coordinates $k_1, k_2 \ldots k_n$ such that $\underline{k}^T \in K_\Gamma$ if and only if $\underline{p}^T \in P_\Gamma$. It was shown in [1], that for state feedback, eq.(9), this is accomplished by an affine mapping

$$\underline{k}^T = [\underline{p}^T \quad 1]\,\underline{E} \qquad (10)$$

where the pole assignment matrix $\underline{E}$ describing the plant is determined by a controllable pair $\underline{A}$, $\underline{b}$ as follows:

Let $\underline{R} = [\underline{b} \quad \underline{A}\,\underline{b} \ldots \underline{A}^{n-1}\,\underline{b}]$

$\underline{e}^T = [0 \ldots 0 \quad 1]\,\underline{R}^{-1}$

Then

$$\underline{E} = \begin{bmatrix} \underline{e}^T \\ \underline{e}^T\underline{A} \\ \vdots \\ \underline{e}^T\underline{A}^n \end{bmatrix} \tag{11}$$

By this affine mapping hyperplanes remain hyperplanes and conic sections remain conic sections. Thus all principal properties of the regions can be studied in the canonical parameter space $\mathscr{P}$. A system $(\underline{A},\underline{b})$ is interpreted as an affine mapping from $\mathscr{P}$-space to $\mathscr{K}$-space.

For each pair $\underline{A}_j$, $\underline{b}_j$, i.e. for each flight condition, a different mapping $\underline{E}_j$ results and the solution set is the intersection of the regions $K_{\Gamma j}$ in $\mathscr{K}$-space. Graphical representation of such regions is easy for n = 2 and possible for n = 3 by computer graphics. For higher system orders the design may proceed stepwise by fixing n-2 gains in each step, also for output feedback some gains are fixed. In the aircraft example n = 3 and $k_3 = 0$, i.e. we are looking at a two dimensional cross-section of the three-dimensional region K_Γ.

By eq.(10) each fixed gain k_i implies a linear relationship $k_i = [\underline{p}^T \quad 1]\,\underline{n}_i$, where $\underline{n}_i$ is the ith column of E. These n-2 linear equations can be used to express the two free gains by coefficients of a second order factor of the characteristic polynomial, which is varied along the boundary in s-plane to produce the boundary in the plane of the two free gains[2]. This tool will be applied to the aircraft example in the next section.

III ROBUSTNESS WITH RESPECT TO FLIGHT CONDITION

The first design objective will be to design an output feedback controller, eq.(4), which meets the nominal pole region requirements at all four flight conditions.

The boundary for flight condition 2 is shown in Fig. 5. On a-b eigenvalues are on the lower natural frequency boundary $\omega_{sp} = 3.5$, on b-c they are on the damping 0.35 lines. At c-a real root boundary takes over: on c-d the actuator eigenvalue is at $\sigma = -70$. On d-e a real short period eigenvalue is at the upper natural frequency limit $\sigma = -12.6$ and for e-a the actuator eigenvalue is at $\sigma = -12.6$. The condition for having no real real root $\sigma = -3.5$ is satisfied in the total region. This region R_{nom2} is bounded by two straight lines c-d and d-a resulting from real root conditions and by the two complex boundary curves a-b and b-c. Note that the boundaries in s-plane are conic sections and thus a-b and b-c are segments of conic section also.

The regions R_{nom1} - R_{nom4} for the other flight conditions were found by mapping the eigenvalue constraints for each flight condition into the k_{Nz} - k_q-plane. These four regions have the intersection R_{nom} shown in Fig. 6. Thus robustness with respect to changing flight conditions can be achieved by static output feedback of the accelerometer and gyro signals. More precisely: All eigenvalues at all four flight conditions are in their prescribed regions in s-plane if and only if the pair k_{Nz}, k_q is chosen in the region R_{nom}.

As an example choose the design point Q_1, i.e. $k_{Nz} = -0.115$, $k_q = -0.8$. The closed loop eigenvalues are given in table 2.

Table 2

FC	Short period eigenvalues		Actuator eigenvalue
	damping	natural frequency	
1	0.94	4.68	- 18.31
2	0.61	9.18	- 37.29
3	0.79	4.63	- 17.78
4	0.55	8.11	- 27.04

The selection of a design point in R_{nom} is a tradeoff, in which the designer learns, which requirements are conflicting. E.g. structural vibrations are most critical in flight condition 2 (high speed, low altitude). They can be reduced by avoiding the vicinity of the $\sigma_2 = -70$ boundary. Low damping is most critical at the supersonic flight condition 4. Damping can be increased by avoiding the vicinity of the $\zeta_4 = 0.35$ boundary. Sluggish responses in landing approach can be avoided by avoiding the vicinity of the $\sigma_1 = -2.02$ boundary. The $\sigma_1 = -7.23$ boundary is only necessary in order to separate actuator and short period poles, the design point may be chosen close to this boundary.

IV ROBUSTNESS WITH RESPECT TO SENSOR FAILURES

As far as stability is concerned, a failure of the accelerometer(gyro) is equivalent to a reduction of k_{Nz} (k_q) from the nominal value to zero or some value in between. Fig. 6 shows that the nominal region does not intersect the axes, thus in the assumed output feedback structure it is not possible to maintain nominal specifications after their failure.

Fig. 6 shows however that a considerable gain reduction inside R_{nom} is admissible, if k_{Nz} and k_q are reduced simultaneously. This can be achieved by replacing the accelerometer measurement N_z by an estimate $\hat{N}_z$, which is produced by a filter from q. It is not necessary that this is a true estimate, e.g. generated by an adaptive observer. It is sufficient, that this is a constant filter connected to q such that the frequency response from u to the filter output $\hat{N}_z$ is an approximation to the frequency response from u to N_z for an average over the four flight conditions. Of course separation does not hold, i.e. we can not take the same pair of feedback gains k_{Nz}, k_q as in the case of accelerometer measurement. However this consideration leads to a structure of the feedback system with a two dimensional signal basis q and $\hat{N}_z$, and a new exact determination of admissible regions in the plane of the two feedback gains k_{Nz}, k_q, see Fig. 7 , can be made.

Both transfer functions from u to N_z and to q have the same denominator, thus the filter has to cancel approximately the zeros in the q-channel and to replace them by the averaged zeros of the N_z channel. Table 3 shows the zeros and gain ratios of the transfer functions at the four flight conditions.

Table 3

Open Loop Zeros and Gain Ratio

FC	MACH	ALTITUDE	q-ZERO	Nz-ZEROS	K_N/K_Q
1	.5	5000'	- .884	-.542±j5.33	.527
2	.85	5000'	-1.57	-.929±j9.12	.536
3	.9	35000'	- .637	-.392±j5.67	.537
4	1.5	35000'	- .826	-.481±j8.05	.577
	AVERAGED VALUES		- .98	-.586±j7.04	.543

Fortunately the gain ratio is almost constant. The filter is then

$$\frac{\hat{N}_z}{q} = 0.543 \frac{s^2 + 1.172s + 49.9}{(s + 0.98)} \frac{10}{s + 10} \qquad (12)$$

The term 10/(s+10) was included to make the filter realizable. The pole at s = -0.98 approximately cancels the q-zero and is therefore weakly controllable from u, i.e. the corresponding closed loop pole will remain in the vicinity of -0.98. This however has little effect on the C^* step responses and is exempted from the pole region requirements.

Note that the corresponding idea to use the accelerometer only and to omit the gyro leads to the inverse filter of eq.(12). Here the approximate cancellation occurs for a complex pair in the vicinity of s = -0.586 ± j7.04, i.e. close to the imaginary axis and no robustness with respect to changing flight condition can be achieved [1].

Fig. 8 shows the intersection of the admissible regions for the four flight conditions. It is seen, that flight conditions 2 and 3 are the critical ones, in the accelerometer feedback case flight conditions 1 and 4 were the most critical ones. However the two feedback gains have the same order of magnitude and the shape and extension of the admissible region still admits a choice of $\underline{k}$ as shown in Fig. 8 , namely

$$\underline{k} = \begin{bmatrix} k_{Nz} \\ k_q \end{bmatrix} = \begin{bmatrix} -0.09 \\ -0.8 \end{bmatrix} \tag{13}$$

such that the pole region requirements are satisfied for $2\underline{k}/3$, $\underline{k}/2$ and $\underline{k}/3$.

The failure detection logic in Fig. 7 decides as follows

a) Three gyros unfailed: $g_1 = g_2 = g_3 = 1/3$

b) Gyro i failed: $g_i = 0$, $g_j = 1/2 \quad j \neq i$.

Before the decision b) has been made, we have a case between $\underline{k}$ and $2\underline{k}/3$. If a second gyro fails after decision b) has been made, we have a case between $\underline{k}$ and $\underline{k}/2$. Only in the unlikely case that a second gyro fails before the first failure has been detected, the gain may be reduced to $\underline{k}/3$. For the two typical cases $\underline{k}$, $2\underline{k}/3$, $\underline{k}/2$ and $\underline{k}/3$ the eigenvalues are given in the appendix. They meet all pole region requirements. Also the C^* step responses have been simulated for the open loop $k_{Nz} = k_q = 0$, Fig. 9, and for the closed loop with $\underline{k}$ and $\underline{k}/2$, Fig. 10. Fig. 11 shows the corresponding elevator deflections δ_e.

V CONCLUSION

It has been demonstrated, that an integrated view of redundancy and handling quality requirements for the control of the short period mode of an unstable fighter plane can lead to a simple fault tolerant control system. It is interesting that the measurement by gyros alone not only saves the cost of additional accelerometers but even has advantages for the control system. It is also interesting that a constant controller can control the aircraft at very different flight conditions. Here it must be noted of course that nice stabilization for several typical stationary flight conditions, i.e. different linearizations of a nonlinear system is only a necessary, not a sufficient condition for the stability of the nonlinear system. The nonlinear system is usually tested in simulations. Also for these simulations it is an advantage if the control system is simple, i.e. does not require gain scheduling and different sensor types and if failure detection is not vital for stabilization.

VI REFERENCES

[1] S.N. Franklin — Design of a Robust Flight Control System, MS Thesis, University of Illinois, Urbana-Champaign, EE Dept., August 1979.

[2] S.N. Franklin, J. Ackermann — Robust Flight Control - A Design Example. To be published.

[3] R.L. Berger, J.R. Hess and D.C. Anderson — Compatibility of Maneuver Load Control and Relaxed Static Stability Applied to Military Aircraft. AFFDL-TR-73-33, April 1973.

[4] Flying Qualities of Piloted Airplanes — MIL-F-8785B (ASG), 7 Aug. 1969.

[5] J. Ackermann — Parameter Space Design of Robust Control Systems, IEEE Transactions on Automatic Control, Oct. 1980.

VII ACKNOWLEDGEMENT

The author gratefully acknowledges the contributions of D. Kaesbauer und N. Franklin.

APPENDIX

1.) Aerodynamic data for eq.(1)

	FC 1	FC 2	FC 3	FC 4
Mach	0.5	0.85	0.9	1.5
Altitude	5000'	5000'	35000'	35000'
a_{11}	- 0.9896	- 1.702	- 0.667	- 0.5162
a_{12}	17.41	50.72	18.11	26.96
a_{13}	96.15	263.5	84.34	178.9
a_{21}	0.2648	0.2201	0.08201	- 0.6896
a_{22}	- 0.8512	- 1.418	- 0.6587	- 1.225
a_{23}	-11.39	- 31.99	-10.81	-30.38
b_1	-97.78	-272.2	-85.09	-175.6

2.) Military specifications for flying qualities, see eq.(3)

Natural frequency (rad/sec)	FC 1	FC 2	FC 3	FC 4
ω_a	2.02	3.50	2.19	3.29
ω_b	7.23	12.6	7.86	11.8

3.) Closed-loop eigenvalues

Complex eigenvalues $s^2 + 2\zeta\omega s + \omega^2$ are written (ζ,ω). The short period eigenvalues are listed first.

Gain	FC 1	FC 2	FC 3	FC 4
$\underline{k}$	(0.60, 4.30) (0.60, 17.2) -0.87	(0.68, 4.63) (0.38, 26.4) -1.63	(0.57, 4.38) (0.64, 16.2) -0.62	(0.65, 5.34) (0.45, 20.9) -0.86
$2\underline{k}/3$	(0.57, 3.86) (0.71, 15.3) -0.86	(0.66, 4.41) (0.47, 22.0) -1.67	(0.52, 3.95) (0.74, 14.5) -0.61	(0.60, 5.47) (0.55, 17.6) -0.87
$\underline{k}/2$	(0.55, 3.47) (0.77, 14.4) -0.85	(0.64, 4.17) (0.54, 19.6) -1.72	(0.50, 3.59) (0.80, 13.8) -0.60	(0.56, 5.54) (0.62, 1.57) -0.88
$\underline{k}/3$	(0.56, 2.86) (0.84, 13.5) -0.83	(0.61, 3.69) (0.64, 17.0) -1.84	(0.48, 3.05) (0.87, 13.1) -0.59	(0.48, 5.53) (0.74, 13.9) -0.90
0 open loop	1.23 - 3.07 -14 -10 - 0.98	1.78 - 4.90 -14 -10 - 0.98	0.56 - 1.87 -14 -10 - 0.98	(0.20, 4.4) -14 -10 - 0.98

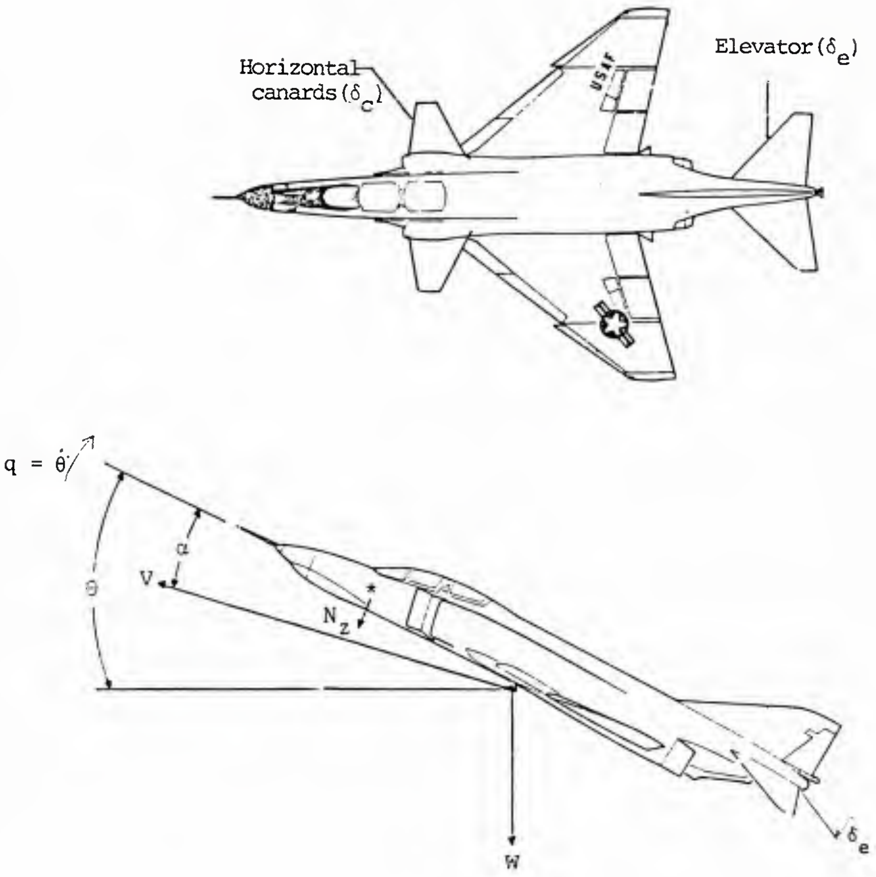


Fig. 1 F4-E with canards

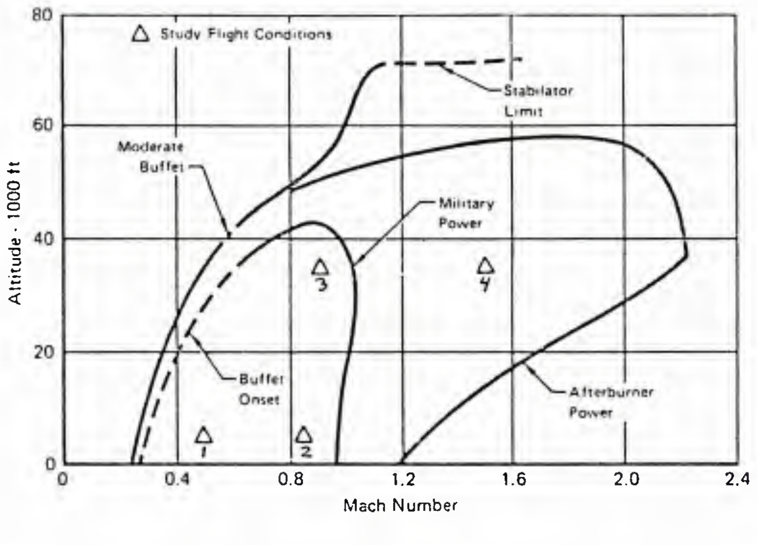


Fig. 2 Flight envelope and operating points [3]

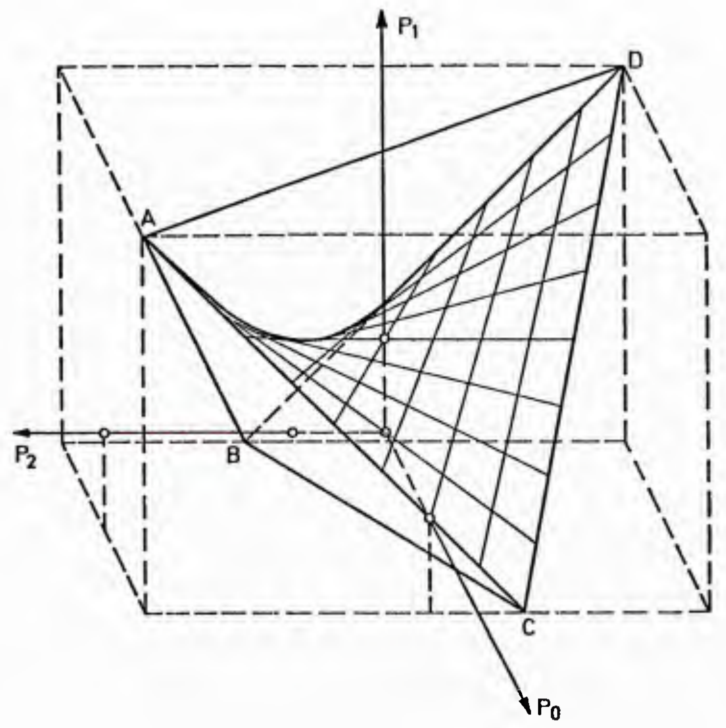


Fig. 4 Zeros of the polynomial
$p_o + p_1 s + p_2 s^2 + s^3 = 0$
are in the unit circle if and only if the coefficients are inside the region bounded by the triangles ABC, BCD (real root boundaries) and by the hyperbolic paraboloid (complex root boundary)

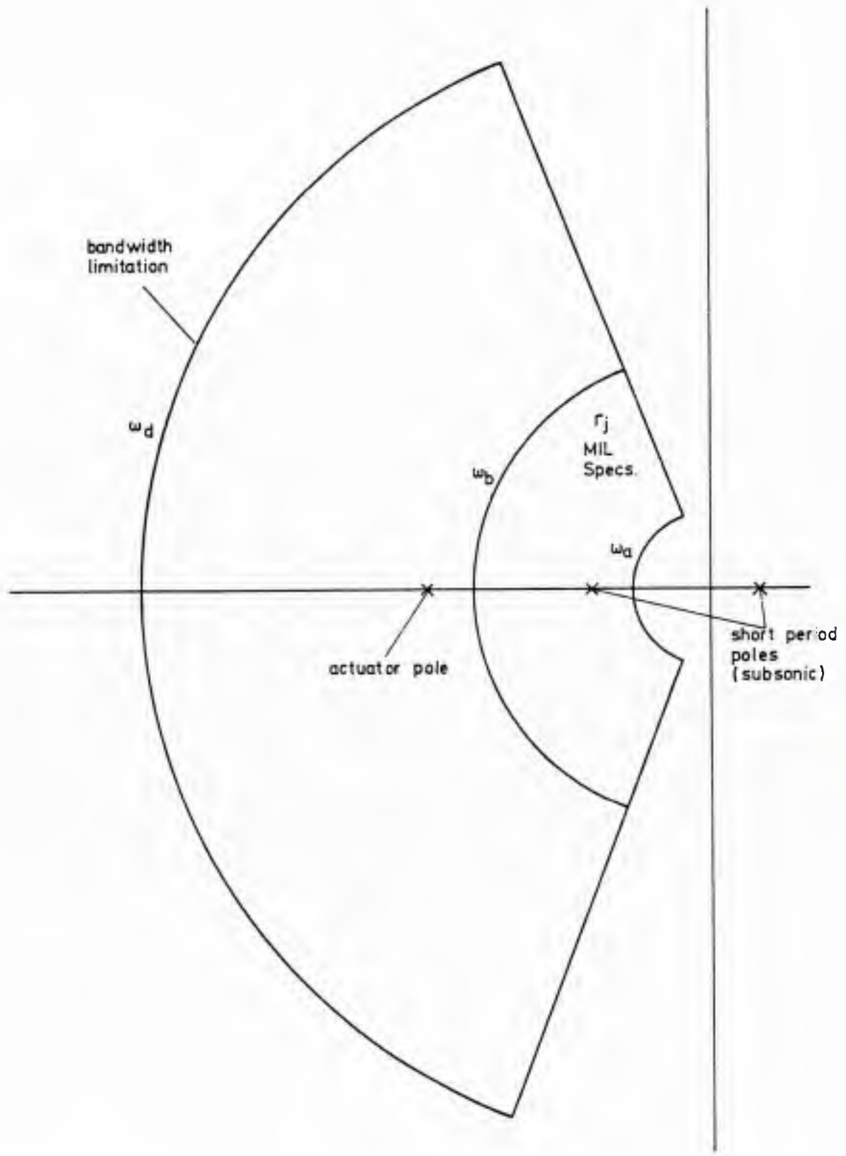


Fig. 3 Required closed loop pole regions

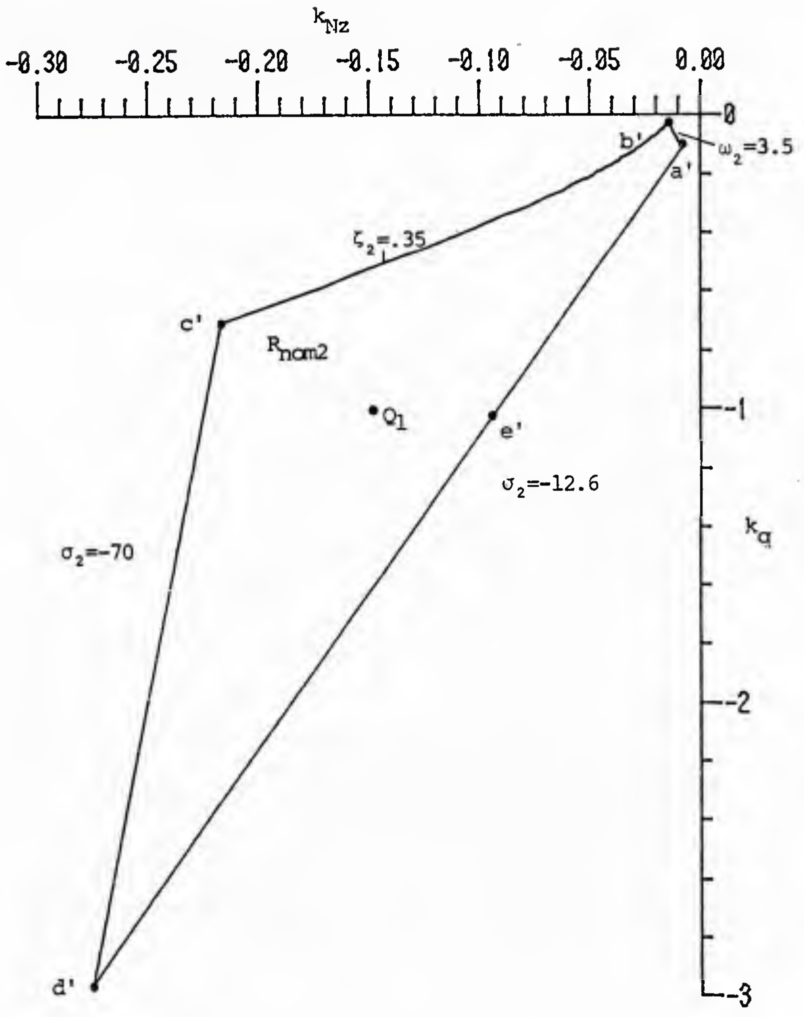


Fig. 5 Admissible region for flight condition 2

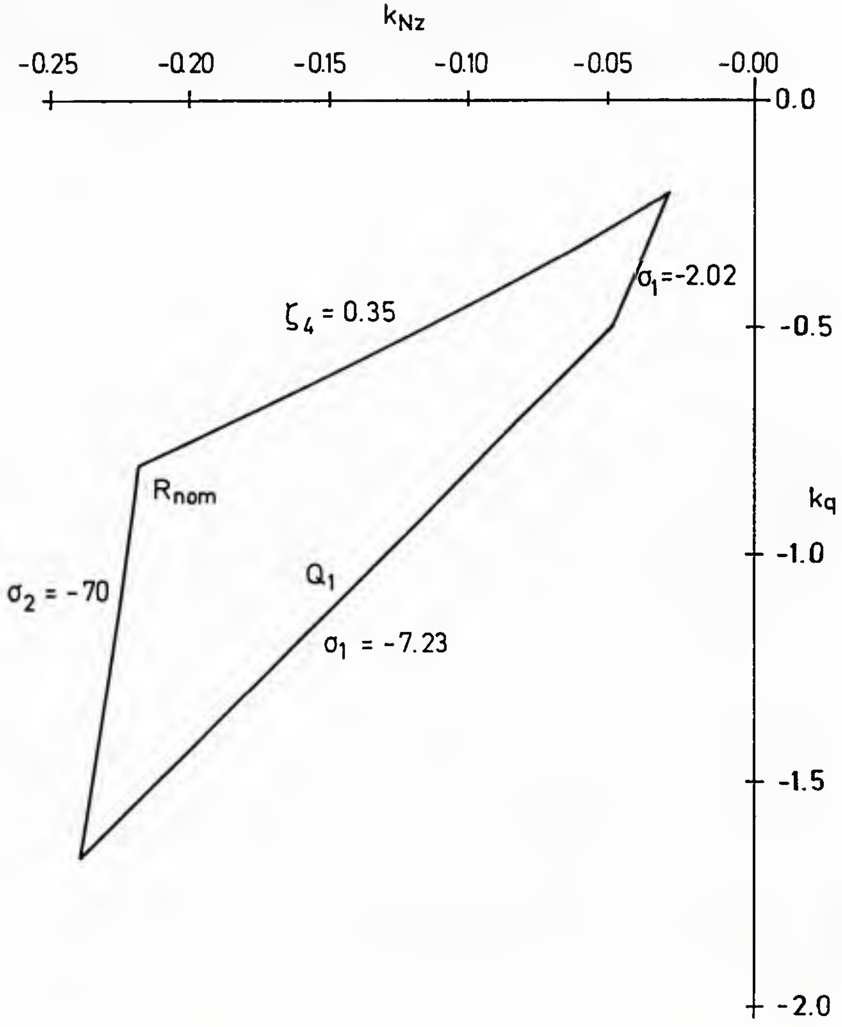


Fig. 6 Intersection of admissible regions for flight conditions 1 through 4

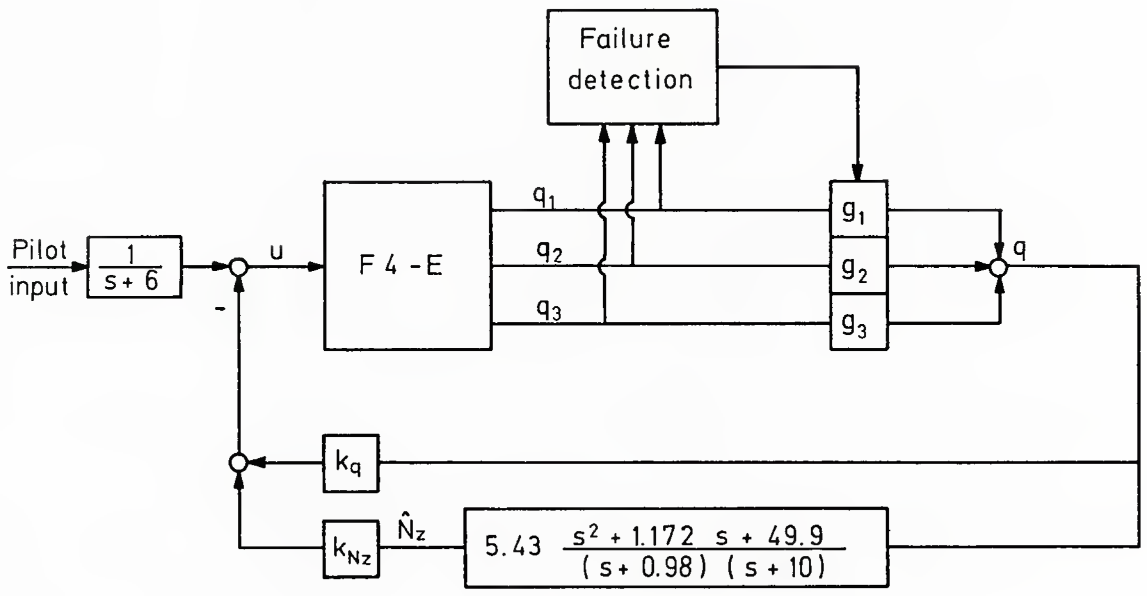


Fig. 7 Robustflight control system with 3 gyros

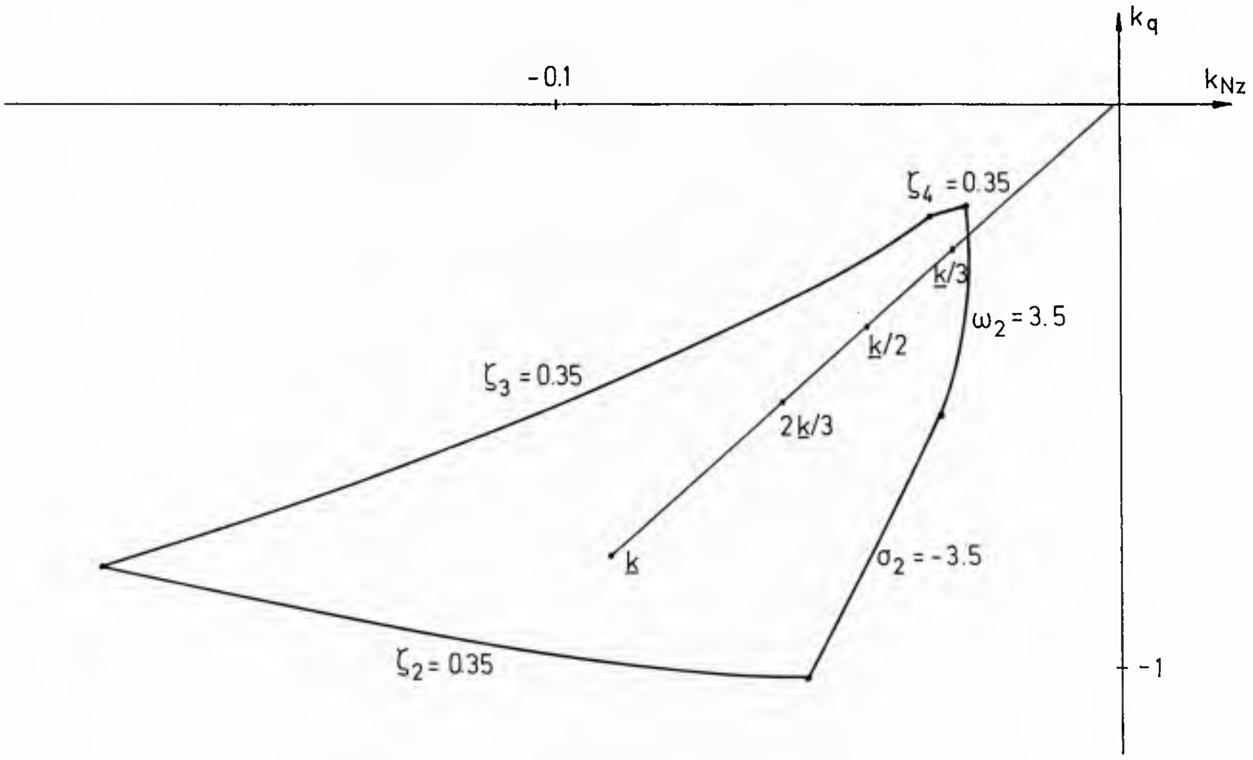


Fig. 8 Admissible region for control system configuration of Fig. 7

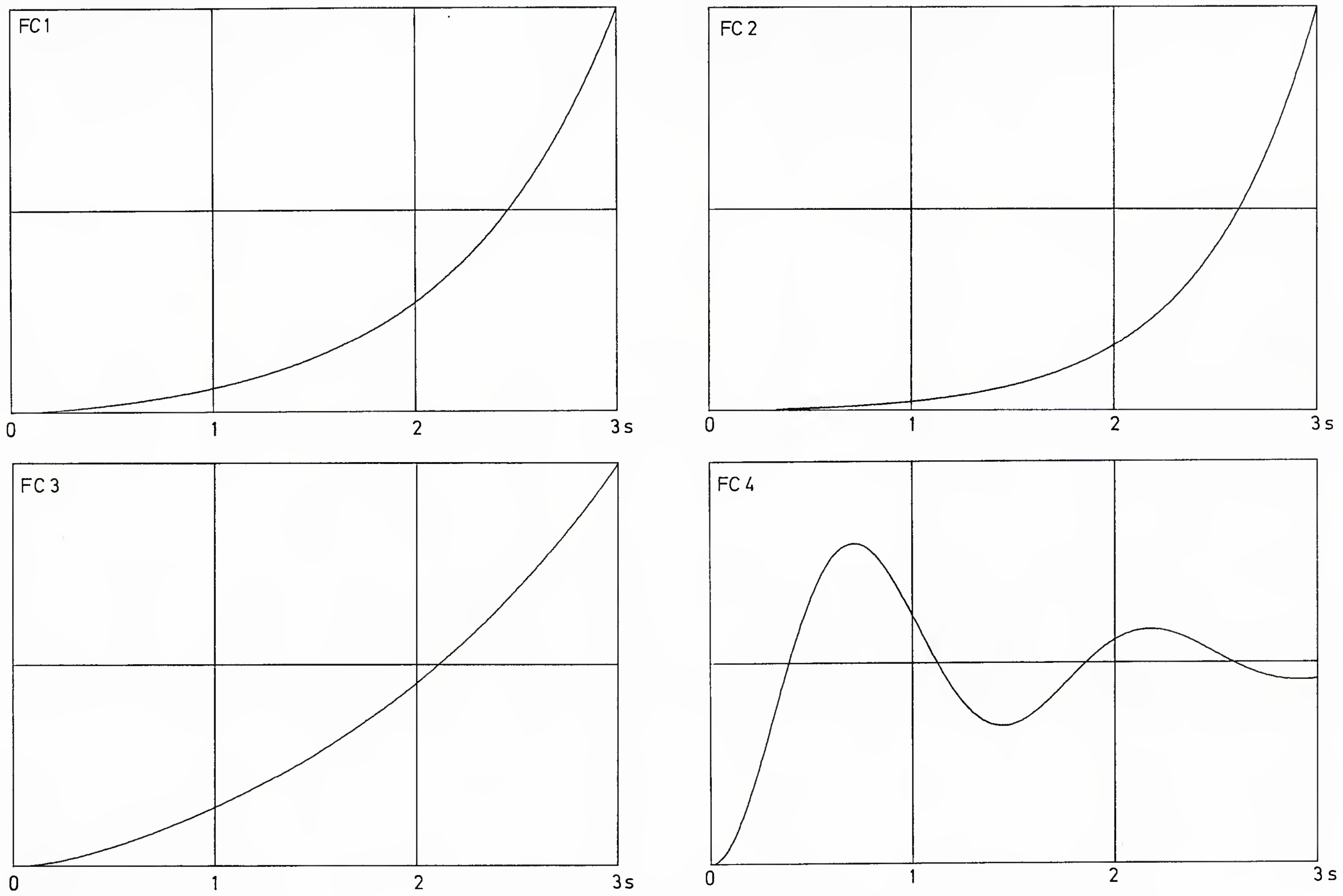


Fig. 9 C^* step responses of the open loop

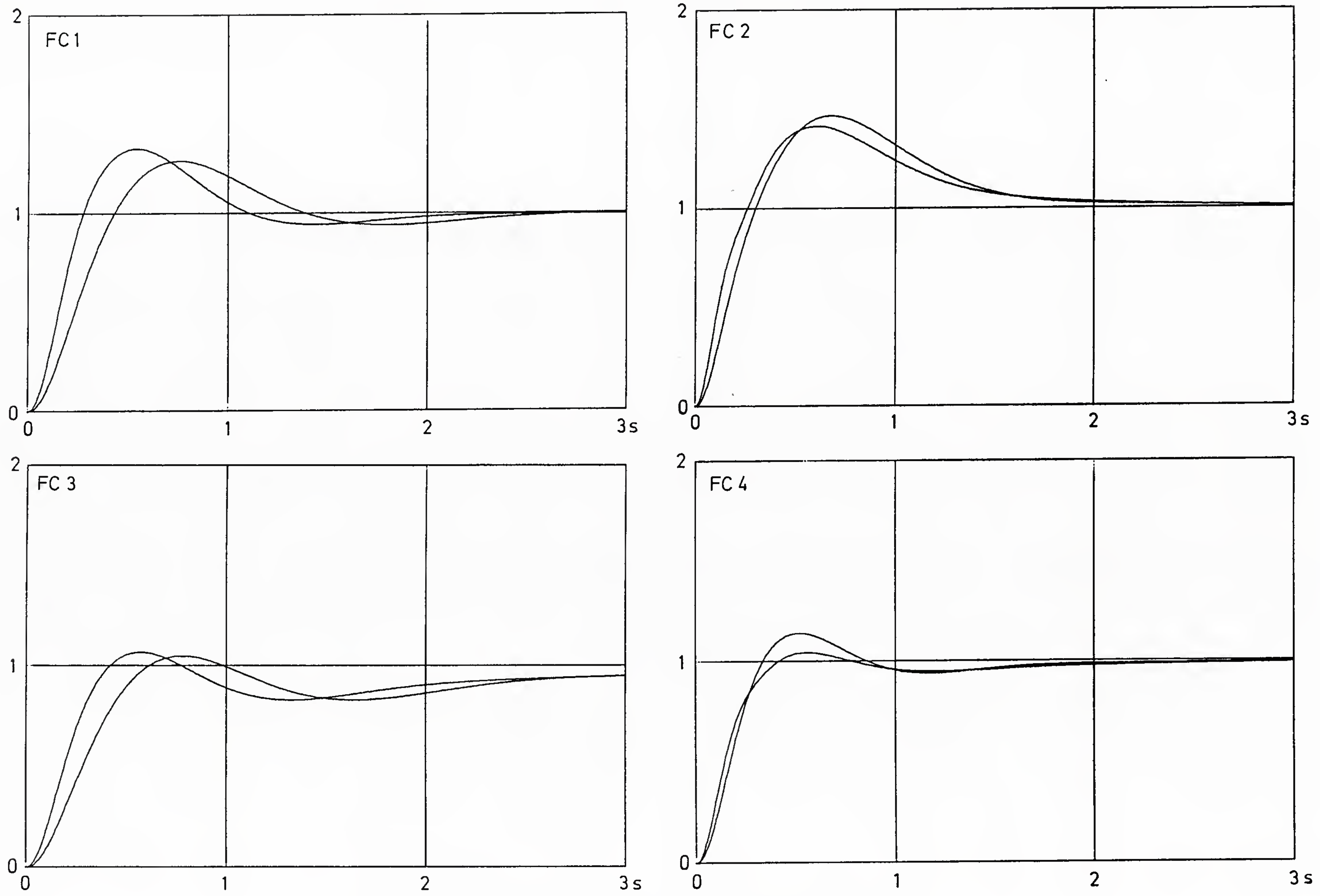


Fig. 10 C^* step responses of the closed loop for k and k/2
(k corresponds to the curve with the steeper initial rise)

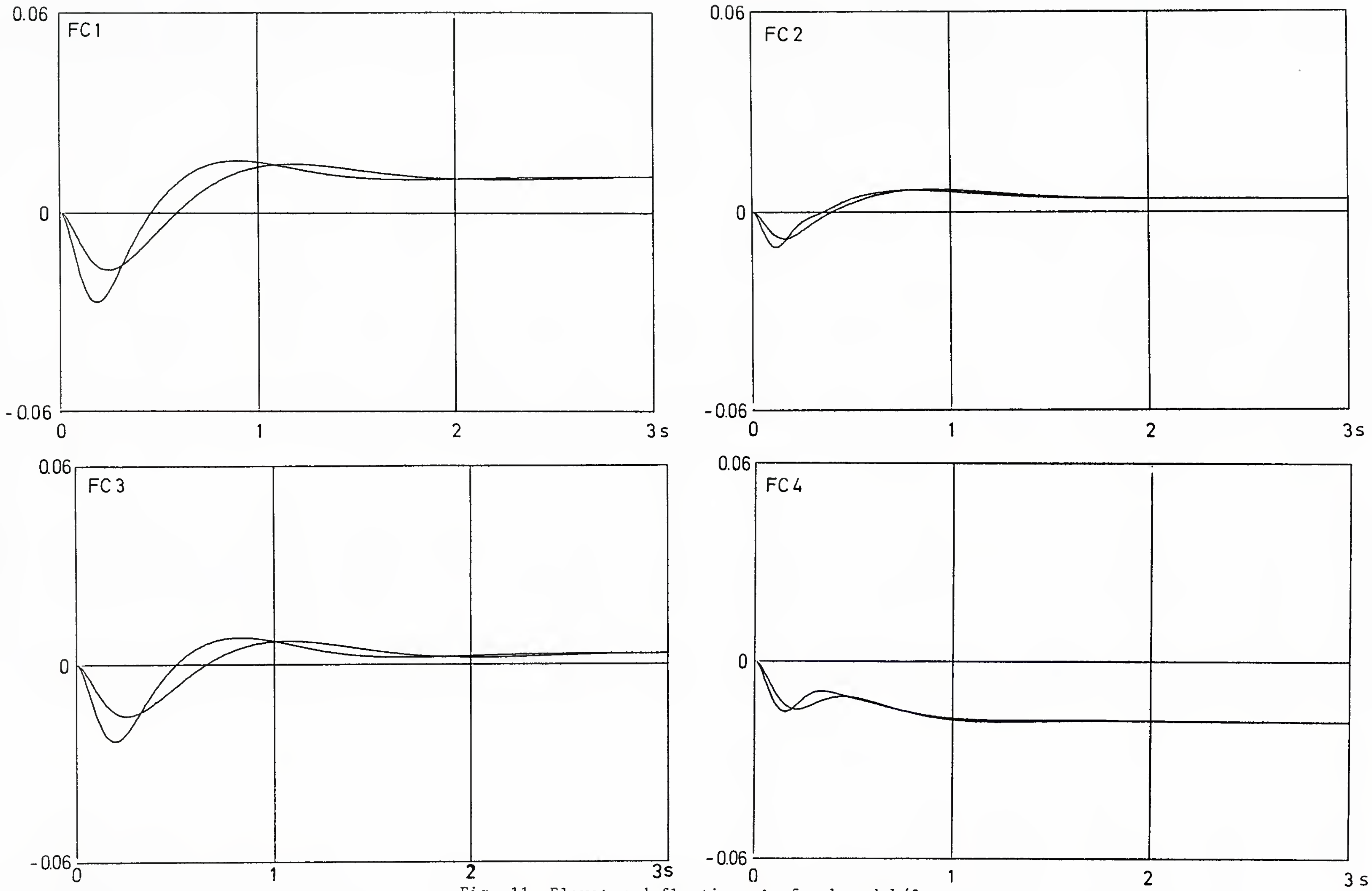


Fig. 11 Elevator deflections δ_e for $\underline{k}$ and $\underline{k}/2$

RELIABILITY MODELING OF FAULT-TOLERANT COMPUTER BASED SYSTEMS

Salvatore J. Bavuso
NASA Langley Research Center
Mail Stop 130
Hampton, Virginia 23665-5225
USA

SUMMARY

Digital fault-tolerant computer-based systems are on the verge of becoming commonplace in military and commercial avionics. These systems hold the promise of increased availability, reliability, and maintainability over conventional analog-based systems through the application of replicated digital computers arranged in fault-tolerant configurations. Three tightly coupled factors of paramount importance, ultimately determining the viability of these systems, are reliability, safety, and profitability. Reliability, the major driver, affects virtually every aspect of design, packaging, and field operations and eventually produces profit for commercial applications or to increased national security for military uses.

The antithesis of promise for the digital computer is, however, the Achilles' heel of the reliability engineer. The utilization of digital computer systems makes the task of producing credible reliability assessment a formidable one. The root of the problem is embodied in the very essence that makes the digital computer such an outstanding device for a host of applications, namely its adaptability to changing requirements, computational power, and ability to test itself efficiently. It is the intent of this paper to address the nuances of modeling the reliability of systems with large state sizes, in the "Markov" sense, which result from systems that are based on replicated redundant hardware and to discuss the modeling of numerous factors which can reduce reliability without concomitant depletion of spare hardware. The diminishing factors are captured by the inclusion of "coverage" parameters that capture in probabilistic measures the effectiveness of system handling of faults. Advanced coverage (fault-handling) models are described and methods of acquiring and measuring parameters for these models are delineated. Some recently measured latent-fault data are also presented.

1. INTRODUCTION

The development of two novel methodologies for the reliability assessment of fault-tolerant digital computer-based systems is reported in this paper: Computer-Aided Reliability Estimation III and Gate Logic Software Simulation. Both technologies were developed to mitigate a serious weakness in the design and evaluation process of ultrareliable digital systems. The weak link is the unavailability of a sufficiently powerful modeling technique for comparing the stochastic attributes of one system against others. Some of the more interesting attributes are reliability, system survival, safety, and mission success.

A long-term goal of the NASA Langley Research Center is the development of this tool. The technology development process is shown in figure 1. Historically, our interest in this subject commenced circa 1971. At that time, two math models were identified as having potential for filling the assessment gap. Figure 1 shows those models as CARE, Computer Aided Reliability Estimation, a computer program generated at NASA's Jet Propulsion Laboratory for application to long-lived, space borne computer systems; and TASRA, Tabular System Reliability Analysis, a computer program due to Battelle Memorial Laboratories for application to the F-111 Pitch Flight Control System (refs. 1 and 2).

The CARE computer program is a very powerful reliability assessment capability for fault-tolerant system concepts that existed in the late 1960's. A major innovation in reliability modeling in CARE was the incorporation of the stochastic concept of coverage due to Roth et al. (ref. 3). Coverage, defined as the conditional probability that a proper recovery occurs if a fault exists, was shown by Bouricius et al. to be a significant factor for achieving high reliability in modular replacement systems (ref. 4). Prior to this consideration, reliability analyses omitted the coverage parameter entirely which caused math models to assume a unity probability of system recovery given a fault occurrence, thereby forcing the reliability predictions to be nonconservative and hence, inaccurate. Although powerful and innovative, the CARE math model suffers from two major deficiencies, its inflexibility to model the emerging multiprocessor-based systems and the lack of a model for computing the coverage parameter.

The TASRA computer program, in contrast to CARE, utilizes the popular "Markov" analysis method which allows a very flexible modeling technique but lacks the vital coverage model as well.

Based on these findings, NASA Langley participated in the codevelopment of CARE II with the Raytheon Company (refs. 5 and 6). The primary objective in creating CARE II was to develop a coverage model to compute coverage for CARE. Figure 2 presents the coverage math model and delineates the factors comprising the coverage computation. Although the CARE II model represents a quantum leap in coverage modeling, CARE II still retains the architectural-description inflexibility of the original CARE system.

A gestation period ensued following the CARE II development that involved Langley in numerous studies, some of which are depicted in figure 1 as square blocks, and the codevelopment of two new reliability assessment methodologies, i.e., CAST (Combined Analytic Simulative Technique) and CARSRA (Computer-Aided Redundant System Reliability Analysis). The coverage impact study determined upper and lower bound values of coverage for a fault-tolerant triplex flight control computer system utilizing state of the art hardware (ref. 7). The CAST study made two important contributions to our program at Langley: it emphasized the potential importance of transient modeling in reliability predictions, and it introduced the notion of combining an analytical approach with computer simulation (ref. 8). By

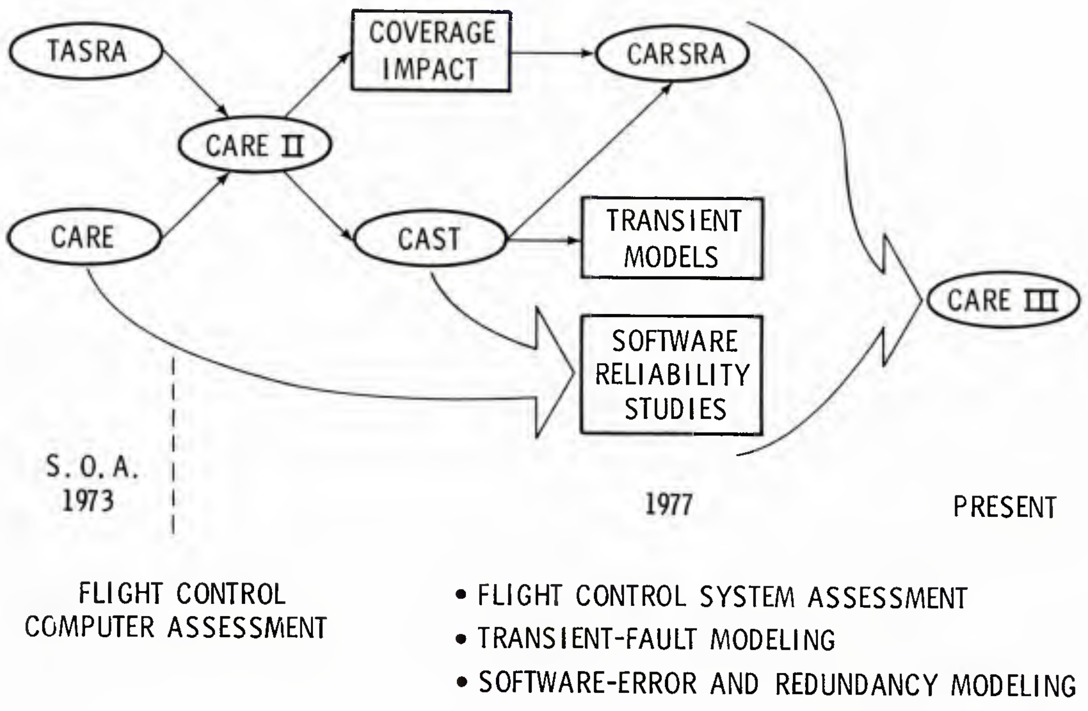

TASRA
COVERAGE IMPACT
CARSRA
CARE II
CARE
CAST
TRANSIENT MODELS
SOFTWARE RELIABILITY STUDIES
CARE III
S.O.A. 1973
1977
PRESENT
FLIGHT CONTROL COMPUTER ASSESSMENT
• FLIGHT CONTROL SYSTEM ASSESSMENT
• TRANSIENT-FAULT MODELING
• SOFTWARE-ERROR AND REDUNDANCY MODELING

Figure 1.- Technological development leading to CARE III.

$$C_X(i,j) = P_i P_{SX}^{\,j-1} P_i' \int_0^\infty \int_0^\infty g_i(\tau) h_i(\tau' - j\tau_{SX}) r_i(\tau, \tau') \, d\tau \, d\tau'$$

WHERE:

- $C_X(i, j)$ = COVERAGE TERM
- τ = DETECTION TIME
- τ' = ISOLATION TIME
- P_{SX} = DEFECTIVE SPARE DETECTION PROBABILITY
- τ_{SX} = SPARE UNIT TEST TIME
- P_i = NON-COMPETITIVE DETECTION PROBABILITY
- P_i' = ISOLATION PROBABILITY ASSOCIATED WITH P_i
- h_i = ISOLATION RATE
- r_i = RECOVERY PROBABILITY
- g_i = COMPETITIVE DETECTION RATE

Figure 2.- CARE II coverage model.

partitioning the modeling of ultrareliable systems in this latter manner, an otherwise intractable problem, using either approach in toto now becomes workable. The CAST concept has become the mainstay of our approach to reliability assessment since the completion of the CAST study, circa 1974.

CARSRA was a spin-off from a Boeing Company study on the design of an Airborne Advanced Reconfigurable Computer System (ARCS) (ref. 9). The development and application of CARSRA was our first comprehensive involvement with the assessment of complicated aircraft flight control systems. The complexity of flight control systems gave rise to the creation of CARSRA in two important areas. Since CARSRA utilizes the Markov approach, a state reduction technique was required, and it became clear that the assessment technique must model stage dependencies in order to assess a variety of system configurations which constitute continued mission success. The most familiar example of the application is system survival. In a redundant fault-tolerant system, there are many system configurations (arising out of system reconfigurations in response to module failures) which will effect the proper system output; however, there are other system configurations that may be of interest in addition to system survival. Boeing, in the ARCS study, defined the term "functional readiness." It is expressed as a time-dependent probability and is applied to missions containing critical subtasks which will either be performed or not be performed, depending on the operational redundancy level at the time of demand. Boeing cites, as an example, an aircraft automatic landing function for which a certain level of hardware redundancy is required before a landing may be initiated in poor visibility and weather conditions. CARSRA also benefited from its predecessors by incorporating a multicoverage parameter capability and an electrical transient modeling capability.

Transient modeling proceeded in two directions: the stochastic estimation of intermittent failures of computer piece-parts and the modeling of the effects of induced analog transients on digital circuitry (refs. 10 and 11). The latter study is ongoing work that relates an analog transient source with a digital system's activity. The form of this relationship will be a stochastic model for input to a system reliability assessment model.

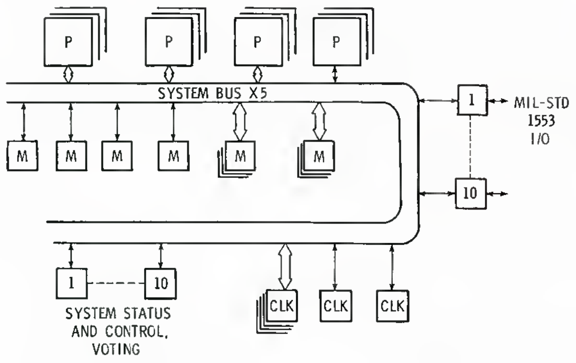
Software reliability studies are an ongoing activity at Langley. The most recent completed study suggests the possibility of estimating software reliability through testing. Although still in the experimental stages, a methodology has been proposed that estimates probability of software error as a function of execution time and test trials (ref. 12).

All of the activities depicted by figure 1 have culminated to form the basis for the development of CARE III. CARE III was codeveloped by Langley and the Raytheon Company and cosponsored by the U.S. Air Force Avionics Laboratory at Wright-Patterson Air Force Base (refs. 13, 14, and 15). The salient features of CARE III are summarized as follows:

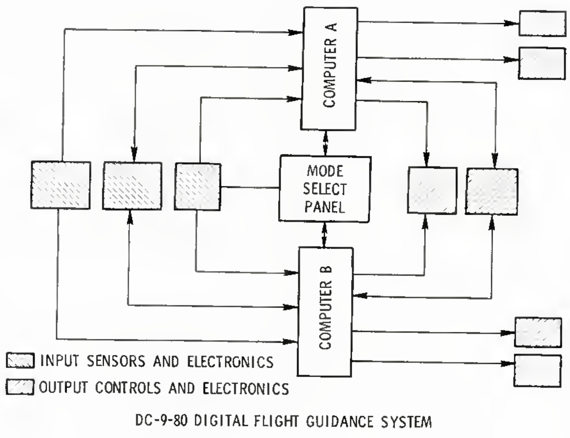
- GENERAL-PURPOSE RELIABILITY ANALYSIS AND DESIGN TOOL FOR FAULT-TOLERANT SYSTEMS
- LARGE REDUCTION OF STATE SIZE
- FAULT-HANDLING MODEL BASED ON PROBABILISTIC DESCRIPTION OF DETECTION, ISOLATION, AND RECOVERY MECHANISMS
- VARIETY OF FAULT AND ERROR MODELS (STATIONARY AND NON-STATIONARY)
    - PERMANENT
    - TRANSIENT
    - INTERMITTENT
    - DESIGN FAULTS
    - SOFTWARE ERRORS
    - LATENT FAULTS
- USER-ORIENTED LANGUAGE FOR DESCRIBING COMPLEX SYSTEM CONFIGURATIONS AND SUCCESS CRITERIA (FAULT TREE)

2. CARE III - A GENERAL-PURPOSE RELIABILITY ANALYSIS AND DESIGN TOOL FOR FAULT-TOLERANT SYSTEMS

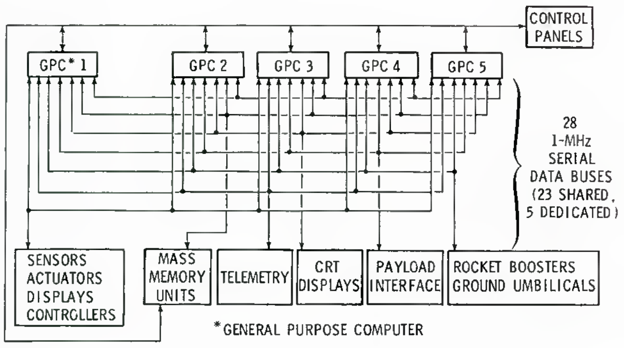
CARE III was designed to model large ultrareliable systems incorporating replicated digital electronic subsystems. Examples of such systems are shown in figure 3 (refs. 16, 17, 18, and 19). The CARE III assessment process is depicted in figure 4 and begins with an architectural description of an ultrareliable system. If that description is based on a conceptual model of the system, then CARE III is used as a design tool; or, if the system is well-defined, CARE III is utilized as an analysis tool. In either case, the analyst generates a set of failure rates and probability density functions for the various failure and error mechanisms he wishes to include in the analysis. A partial list of failure and error models is delineated in figure 5. Inclusion of any of these models will necessarily lower the system reliability estimate. The need to include a model is of course a function of the architectural structure, its fault-handling mechanisms, and the magnitudes of the parameters in the model. The large choice of failure and error models is provided to increase the realism and credibility of the analysis. The models are user options in CARE III and may be omitted at the discretion of the analyst. Usually he will omit certain models after determining that they have a minimal effect. Some of this modeling information is used to define the system fault-handling model(s) required as a user input, depicted by the state diagram in figure 4. The remainder of the failure and error modeling data is entered as Fortran NAMELIST statements for batch computation or the CARE3MENU user-friendly interface program may be used for interactive input (ref. 20).



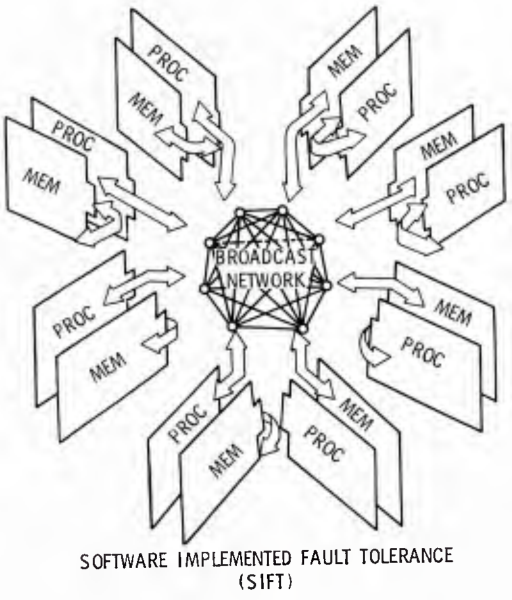
(a) Fault-tolerant multiprocessor.


COMPUTER A
MODE SELECT PANEL
COMPUTER B
▨ INPUT SENSORS AND ELECTRONICS
▨ OUTPUT CONTROLS AND ELECTRONICS
DC-9-80 DIGITAL FLIGHT GUIDANCE SYSTEM

(b) DC-9-80 digital flight guidance system.


CONTROL PANELS
GPC* 1
GPC 2
GPC 3
GPC 4
GPC 5
28 1-MHz SERIAL DATA BUSES (23 SHARED, 5 DEDICATED)
SENSORS ACTUATORS DISPLAYS CONTROLLERS
MASS MEMORY UNITS
TELEMETRY
CRT DISPLAYS
PAYLOAD INTERFACE
ROCKET BOOSTERS GROUND UMBILICALS
*GENERAL PURPOSE COMPUTER

(c) Space Shuttle system



(d) Software implemented fault tolerance (SIFT).

Figure 3.- Highly reliable fault-tolerant systems to which CARE III is applicable.

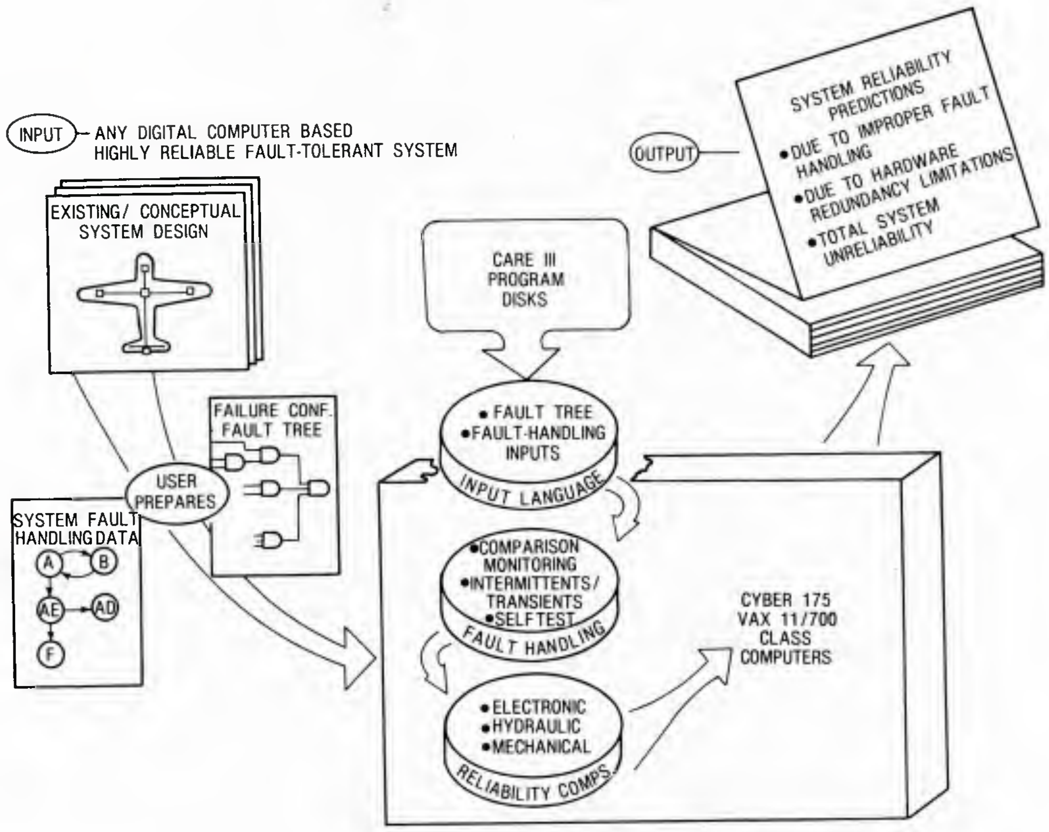

INPUT
ANY DIGITAL COMPUTER BASED HIGHLY RELIABLE FAULT-TOLERANT SYSTEM
EXISTING/ CONCEPTUAL SYSTEM DESIGN
FAILURE CONF. FAULT TREE
USER PREPARES
SYSTEM FAULT HANDLING DATA
A
B
AE
AD
F
CARE III PROGRAM DISKS
• FAULT TREE
•FAULT-HANDLING INPUTS
INPUT LANGUAGE
•COMPARISON MONITORING
•INTERMITTENTS/ TRANSIENTS
•SELFTEST
FAULT HANDLING
• ELECTRONIC
• HYDRAULIC
• MECHANICAL
RELIABILITY COMPS
CYBER 175
VAX 11/700
CLASS COMPUTERS
OUTPUT
SYSTEM RELIABILITY PREDICTIONS
•DUE TO IMPROPER FAULT HANDLING
•DUE TO HARDWARE REDUNDANCY LIMITATIONS
•TOTAL SYSTEM UNRELIABILITY

Figure 4.- CARE III user process.

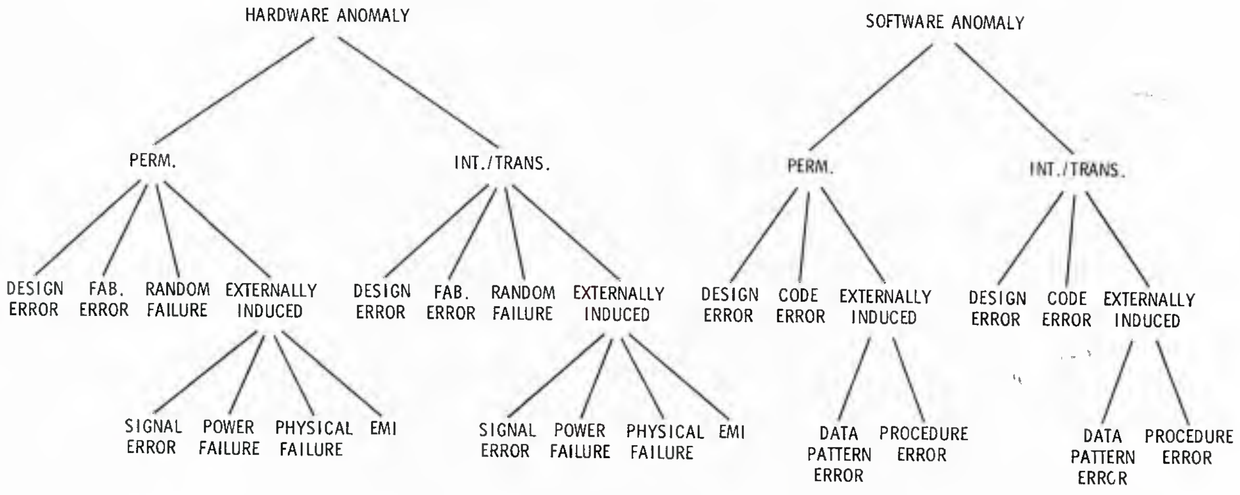


Figure 5.- Delineation of hardware and software failure and error models.

Another important step in setting up CARE III input is the generation of the system configuration and success criteria. Fault-tolerant systems are usually designed to have many hardware and functional combinations that enable proper system operation. CARE III uses the powerful fault tree language to describe system failure configurations. In large systems, the number of success combinations can be very large, and for this reason CARE III uses the "unsuccess" or failure combinations instead. In a properly designed system, the number of failure combinations should be considerably less than the number of system success combinations, thus easing the computational task. The fault tree language provides an excellent medium for delineating the system failure combinations.

Referring again to figure 4, the user-prepared data are initially processed by the CARE III input subprogram, CAREIN, shown as the upper disk. It is essentially composed of the fault tree language program. The second CARE III subprogram, COVRGE, processes the fault-handling model data and puts them into the form required by the third subprogram, CARE3, which performs the reliability computations. CARE III is written entirely in the ANSI standard Fortran 77 language and currently executes on the Digital Equipment Corporation 11/700 series computers, the Control Data Corporation CYBER 170 series computers using the Fortran V compiler, and the International Business Machines 308X series computers. The CARE

III output data take two forms, graphical or tabular. In either case, the outputs of most interest are the total system reliability or system survival as a function of time and two vital components: probability of system failure due to hardware redundancy limitations (depletion of spares) and probability of system failure due to improper fault handling. In ultrareliable systems, the latter factor is the predominant cause of system failure (ref. 20).

An example of the CARE III assessment process (batch input) is given by figures 6 and 7. Figure 6 is a sketch of an ultrareliable fault-tolerant multiprocessor composed of 10 memory-processor pairs which communicate with each other over 5 individuals bus lines shown in the chart as a solid bus line. This system survives if at least 2 computers and 2 buses are operational. The analyst wants to compute the probability of system survival at 10 hours of mission time for this multiprocessor system and the

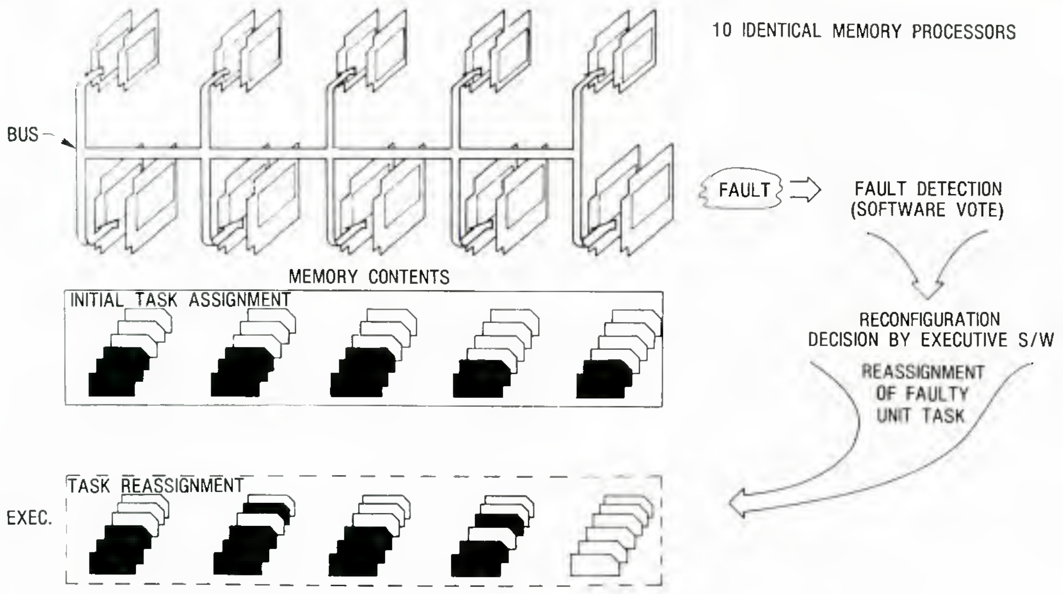
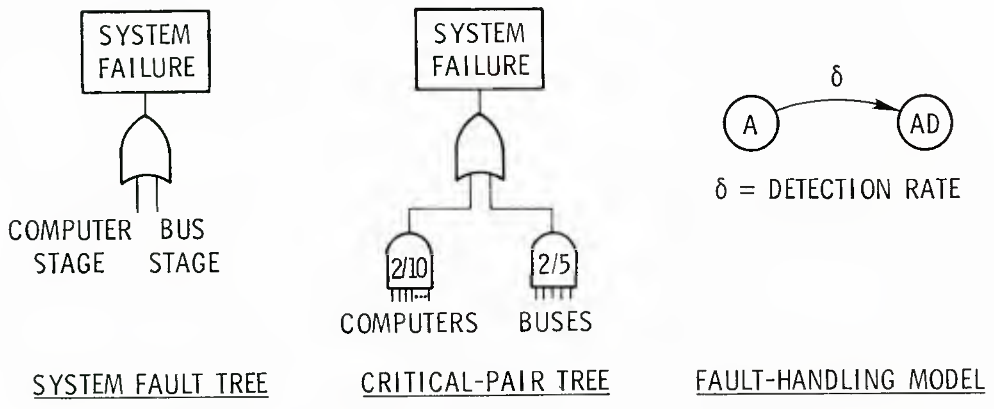

10 IDENTICAL MEMORY PROCESSORS
BUS
FAULT
FAULT DETECTION (SOFTWARE VOTE)
MEMORY CONTENTS
INITIAL TASK ASSIGNMENT
RECONFIGURATION DECISION BY EXECUTIVE S/W
REASSIGNMENT OF FAULTY UNIT TASK
TASK REASSIGNMENT
EXEC.

Figure 6.- Ultrareliable fault-tolerant multiprocessor



```
$FLTTYP NFTYPS = 1, DEL = 3.6E2$
$STAGES NSTGES = 2, N = 10, 5, M = 2, 2, IRLPCD = 4, RLPLOT = T$
$FLTCAT NFCATS = 2*1, JTYP(1, 1) = 1, JTYP(1, 2) = 1,
        RLM(1, 1) = 1.0E-4, RLM(1, 2) = 1.0E-5$
$RNTIME FT = 10. $
SYSTEM FAULT-TREE
1 2 3 3
3 0 1 2
CRITICAL-FAULT PAIRS
1 15 16 18
1 1 10
2 11 15
16 2 1 2 3 4 5 6 7 8 9 10
17 2 11 12 13 14 15
18 0 16 17
```

Figure 7.- CARE III input for ultrareliable fault-tolerant multiprocessor.

probability of system failure at 10 hours due to spare hardware depletion and due to improper system-fault handling. For this simple illustrative example, the analyst creates two fault trees and a state diagram for system-fault handling as depicted in figure 7.

The SYSTEM FAULT TREE describes the system stage configurations that cause system failure resulting from depletion of hardware redundancy. The computer stage is comprised of up to 10 computers, each having an identical failure rate. The bus stage is composed of up to 5 buses, each having and identical failure rate, most likely different from that of a computer. The OR gate in the SYSTEM FAULT TREE means that the system fails if a computer stage fails or a bus stage fails. A computer stage fails if less than 2 computers are operational, and a bus stage fails if less than 2 buses are operational. These conditions are described in the line beginning with "$STAGES". This statement is a Fortran NAMELIST statement. It says there are two stages (NSTGES=2), i.e., 10 computers and 5 buses (N=10,5), and the minimum for stage survival is 2 for each stage (M=2,2). The remainder of the line describes the form of output data requested. The fault tree description for CARE III input is shown under the heading, SYSTEM FAULT-TREE. It describes the gate interconnections and the types of gates. There is another cause of system failure that is implicit in the SYSTEM FAULT TREE and that is system failure due to single-fault failures in either stage. Details of this model are discussed later.

A unique modeling capability of CARE III is the incorporation of the effects of synergistic pairs of failures. In fault-tolerant systems, the system could contain many undetected (latent) failures which individually would not cause system failure; however, certain groupings of failures that coexist may bring the system down. The CRITICAL-PAIR TREE enables the analyst to specify the conditions under which synergistic paired failures cause system failure. For this case, any 2 latent computer failures out of 10 computers or any 2 latent bus failures out of 5 buses cause system failure. For a more accurate analysis, one would usually specify which paired failures cause system failure. The CRITICAL-PAIR TREE is described by the data listed under the heading, CRITICAL-FAULT PAIRS (fig. 7).

The next step in the CARE III input process is the description of the FAULT-HANDLING Model. This simple state model is composed of two system states, active (A) and active detected (AD). The active system state is entered when a module failure occurs. It is an undetected or latent state. If a fault detector is employed, is the rate at which failures are purged from the system. CARE III assumes that if the system enters the active detected state and it has spare hardware, it will reconfigure out the faulty module and the system will recover. Note, as δ increases, the probability of synergistic failures occurring diminishes since there will be fewer latent failures present. In figure 7, line 1, $FLTTYP, shows that there is one fault model (NFTYPS=1) and defines the value of as 3.6×10^2 detections per hour. Line 3, $FLTCAT, simply links failure rates denoted as RLM arrays to stages (JTYP). Line 5, $RNTIME, specifies a flight time of 10 hours.

CARE III input data for this example system begins with the statement $FLTTYP and includes all the statements that follow in figure 7. The CARE III output includes the total system probability of failure, the system probability of failure due to improper fault handling, and the probability of system failure due to depletion of spares.

2.1 User-Oriented Language for Describing Complex System Failure Configurations (Fault Tree)

The multiprocessor example made use of a trivial application of the CARE III fault tree language. A better example emphasizing the power of the fault tree input is given by figures 8 and 9. Figure 8 shows a block diagram of a proposed fault-tolerant flight-control system. Of particular interest is the pitch augmentation stability (PAS) short cycle function. The system fault tree for this function is presented in figure 9. This tree illustrates that not only hardware redundancy can be represented but functional redundancy as well. The elevator math model is functionally redundant to the secondary actuators. The melding of hardware and functional redundancy is a common practice in aircraft design. The proper entry of this fault tree into CARE III with the necessary failure rate and fault-handling data would yield a prediction of the probability of loss of PAS function as a function of mission time. Figure 9 is read as follows. An output from logic OR gate 212 constitutes loss of PAS function which can occur if an output from OR gate 211 occurs, or if an output from gate 210 occurs, or both. Gate 210 yields an output if at least 3 out of 4 secondary actuators or actuator function (elevator math model) fail. Secondary actuator A will fail if computer A fails, or actuator A fails, or both. A similar description can be used to delineate failures due to loss of computation or loss of sensors.

2.2 Fault-Handling Model Based on Probabilistic Description of Operative Detection, Isolation, and Recovery Mechanisms

In figure 7, a simple fault-handling model of two states was described. CARE III has both a single-fault model and a double-fault model for coincident paired failures. The single-fault model is given in figure 10 and is shown in the dashed box. For illustrative purposes, three additional states have been added so that the state diagram is a Markov model of a two-unit system. Initially, the system is in the state 0 and has experienced no failures. When a module failure occurs, the module enters state A, the active latent state, given by the arrival density, $\lambda(t)$. Depending upon the nature of the failure, i.e., permanent, transient, intermittent, etc., the fault-handling model would be defined differently. For example, if the failure were intermittent, $\lambda(t)$ would be the probability density function (PDF) for the arrival of an intermittent, and states A and B define the intermittent model where α and β are constant transition rates into and out of state B. When the module is in state B, the benign state, the failed unit appears to have healed itself; i.e., the manifestation of the failure, a fault, vanishes. However, when the failed manifestation is once again resumed (the fault reappears), the module enters state A where the failure looks like a permanent failure. It could be detected by a self-test program with PDF $\delta(t')$, and the system would enter state A_D, the active detected state. Given that a spare exists, the system will purge the faulty unit and switch in the spare (dashed arc to state 1). Or while in the active state, the fault could generate errors with PDF $\rho(t')$. The module then will enter the A_E, active error state. The intermittent failure could manifest its intermittent state again so the module would then enter state B_E, the benign error state. Although the failure is benign, the error may not be benign and may cause system failure which is denoted by the B_E to F transition $[(1-c)\varepsilon(\tau)]$. The error detection

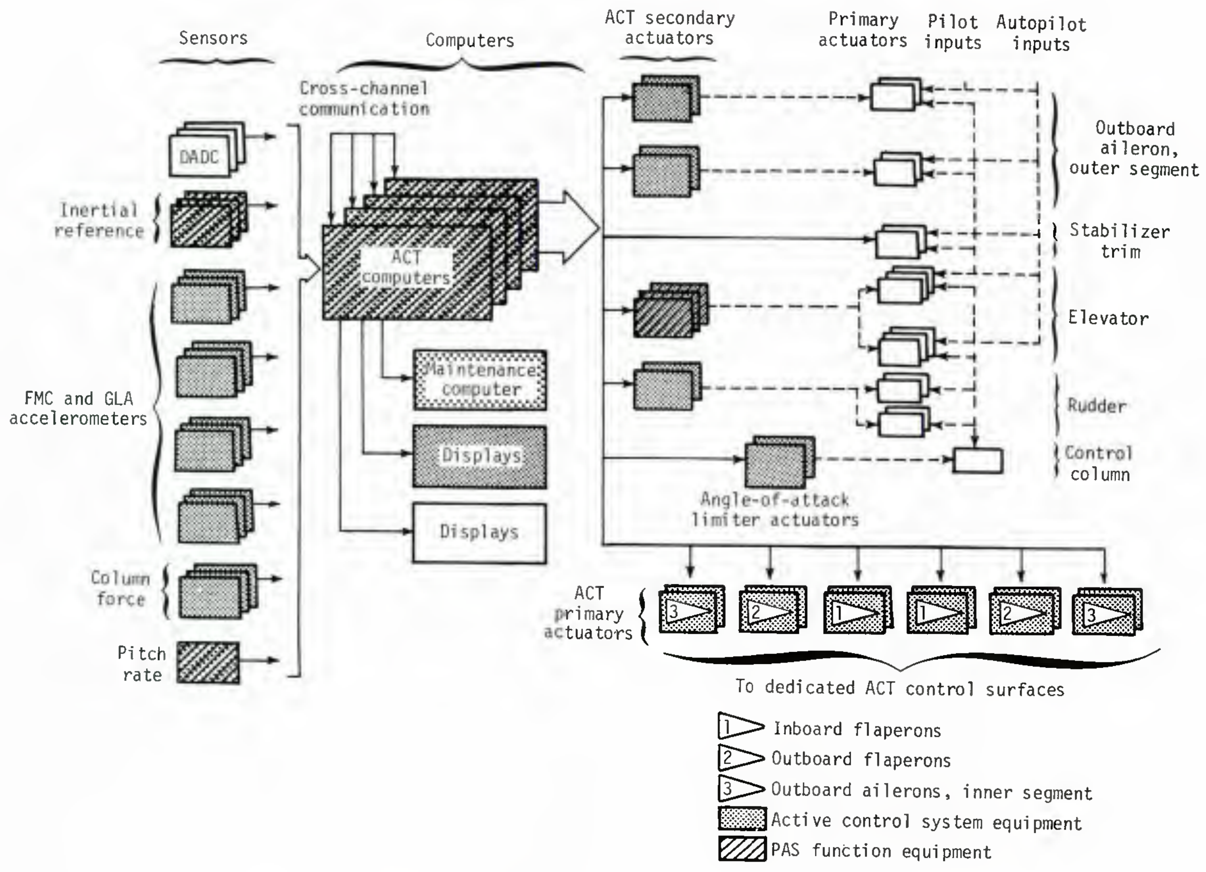


Figure 8.- Fault-tolerant flight control system - pitch augmented stability function (PAS).

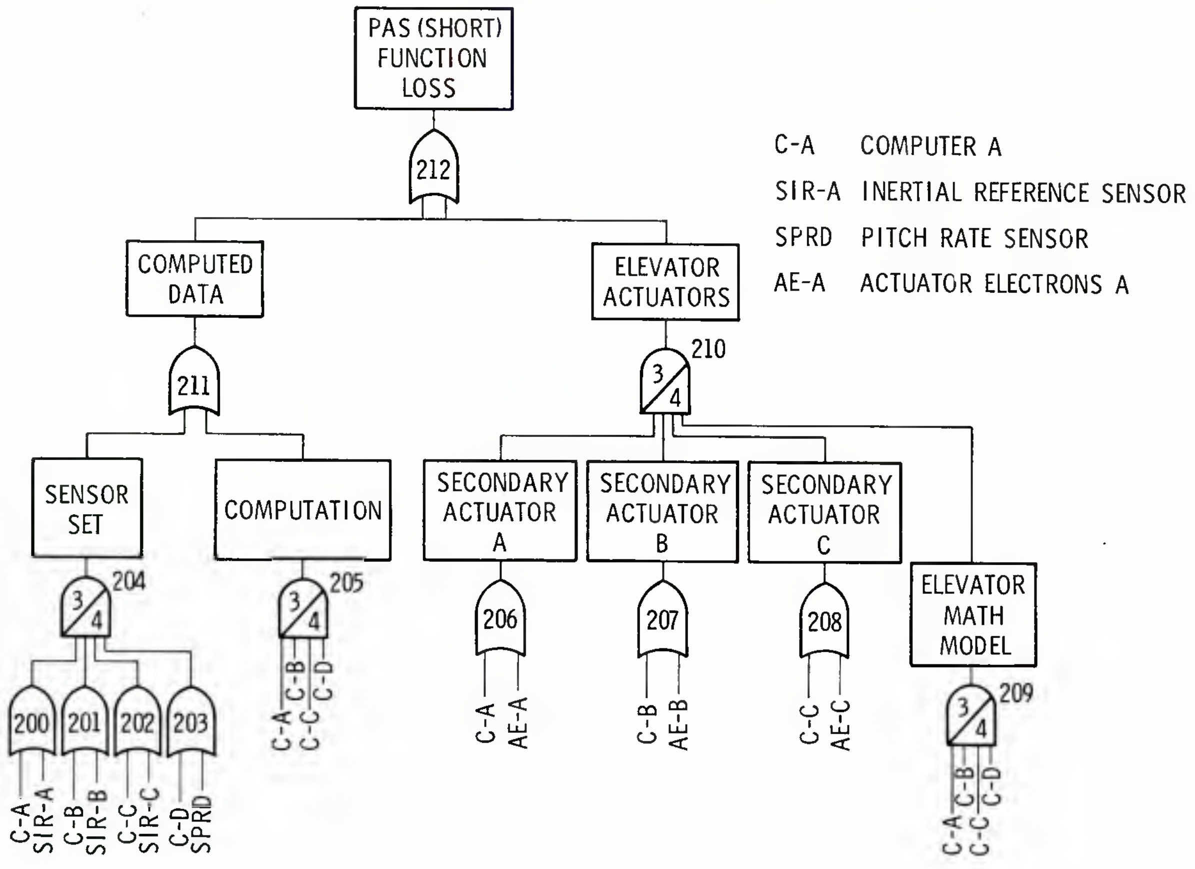


Figure 9.- CARE III fault tree input for PAS function.

density is $\varepsilon(\tau)$, and 1-C is the proportion of errors from which the system is unable to recover. While in state B_E, the error could be detected and corrected. In this event, the module enters state B_D

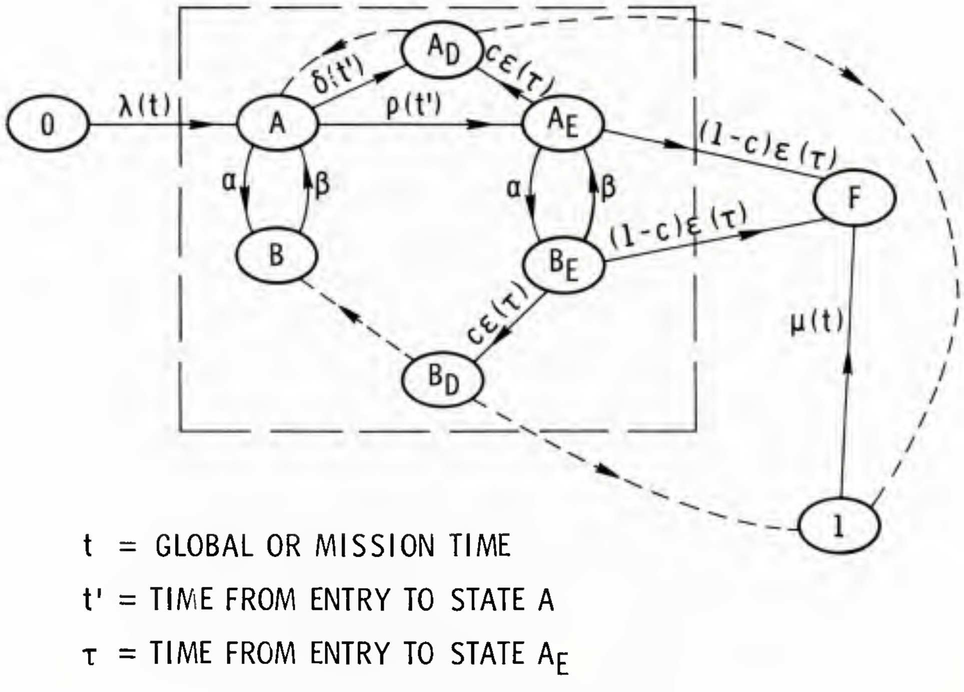

0
λ(t)
A
δ(t')
A_D
cε(τ)
p(t')
A_E
(1-c)ε(τ)
F
α
β
B
α
β
B_E
(1-c)ε(τ)
cε(τ)
μ(t)
B_D
1
t = GLOBAL OR MISSION TIME
t' = TIME FROM ENTRY TO STATE A
τ = TIME FROM ENTRY TO STATE A_E

Figure 10.- Overall reliability model of two-unit system.

(benign detected) by transition $c\varepsilon(\tau)$. At this point, the system may choose to do nothing further with the detected and corrected error and so move to the benign state, or the system amy choose to reconfigure out the module containing the error and, therefore, move to state 1. The dashed arcs are instantaneous transitions. The other transition out of state A_E is to state F, the single point failure transition $[(1-c)\varepsilon(\tau)]$. This transition is similar to the B_E to F transition. In a well-designed fault-tolerant system, $(1-c)\varepsilon(\tau)$ should be near zero. If $\lambda(t)$ is the PDF for the arrival of a transient, α would be set to a value greater than zero and β would be equal to zero. The PDF $\lambda(t)$ for the arrival of a permanent failure would be defined so that $\alpha = \beta = 0$. The dashed arc going from state A_D to A enables the analyst to include the effects of the system decision that the detected fault which took the module from state A to A_D was, in fact, a transient. In this regard, the system would not reconfigure out a non-failed module. A judicious choice of values for the single-fault model affords the analyst a wide range of models. A different fault model may be assigned to each stage, or several models may be assigned to a given stage to cover the effects of different failure mechanisms such as transients, intermittents, and permanent failures.

The reader will note that the reliability model in figure 10 has three measures of time associated with it which necessarily makes the model a semi-Markov process. This added complexity is required because the behavior of the system is dependent on the onset of the various fault-behavior events.

2.3 Large Reduction of System State Size

The capability thus described comes at no small computational price if state of the art techniques are employed. In fact, if one were to utilize the popular "Markov" modeling technique on a nontrivial system such as the flight control system shown in figure 11 (which is composed of 22 stages and 64 reconfigurable modules) coupled with a reasonable set of failure and error models (some of which are delineated in fig. 5), the number of system states would be on the order of millions. For each state, a linear differential equation is formed. Clearly the solution of millions of differential equations is computationally intractable if not impossible with today's technology. But CARE III was designed to assess these types of systems. How does it do it?

To understand the CARE III state reduction method, it is expedient to first examining how the state size buildup occurs in the Markov method. A Markov state is described as an ordered n-tuple. The components of the n-tuple contain information about the number of failed reconfigurable modules in the system plus system fault-handling information for each module and fault type (permanent failure, transient, etc.). For the system shown in figure 11, the n-tuple has a minimum of 22 components, i.e., one for each stage. For each stage, additional n-tuple fault-handling components are added to describe the number of failed modules that are system detected, the number that are identified with a reconfigurable module, and the number that have been recovered. A set of fault-handling components is included in the n-tuple for each type of failure, e.g., transient, permanent, and intermittent. The total number of n-tuple components becomes very large. The product of the n-tuple components gives the number of possible system states; although in practice, the actual number is less than this but still very large. In contemporary practice, tractable analyses are accomplished by making numerous assumptions about the system to reduce the state size to the order of 1000. CARE III, on the other hand, retains a considerable amount of detail without the burden of unmanageable state sizes. This feat is accomplished in CARE III by separating fault-handling information from the fault occurrence, i.e., information about the number of failed units. Each model is worked separately to a point then recombined (ref. 21). An example of this state reduction is depicted by figures 10 and 12. When CARE III processes the fault-

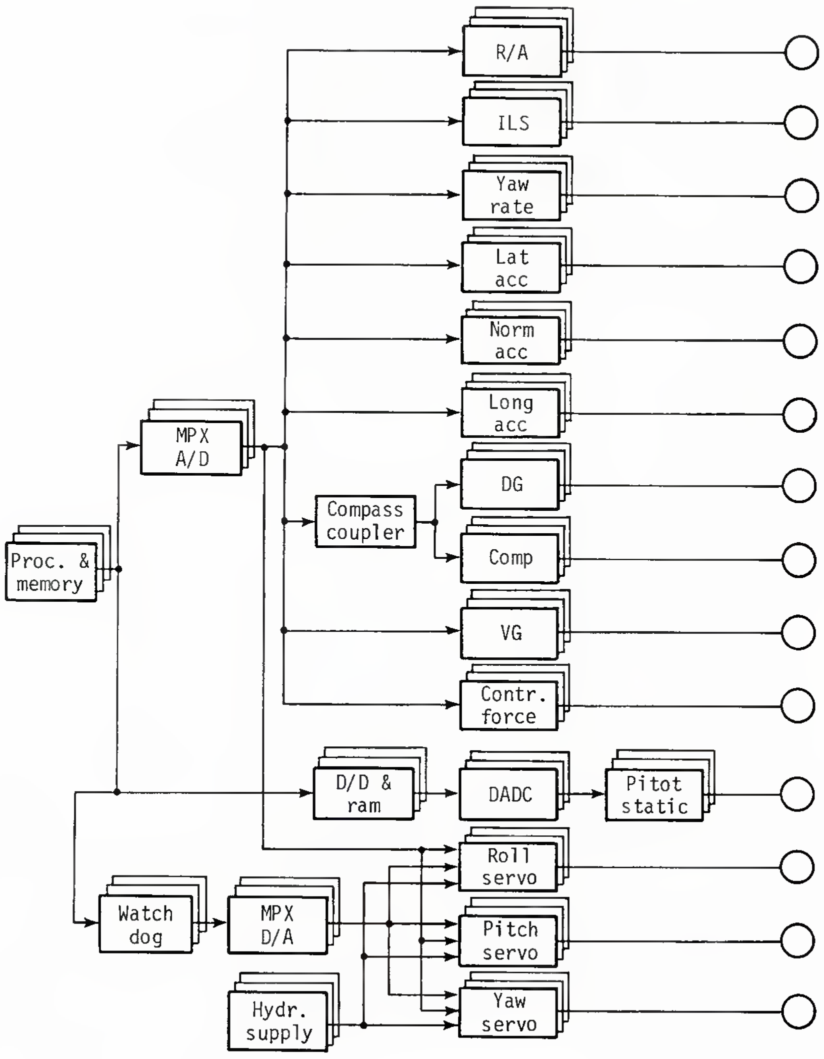


Figure 11.- Advanced reconfigurable flight control system.

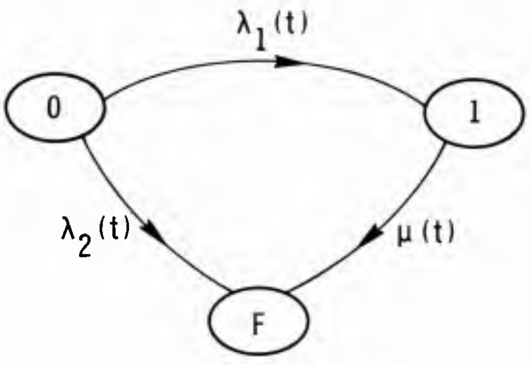


Figure 12.- Aggregated reliability model for a two-unit system.

$$P_l(t) = e^{-\int_0^t \lambda_l(\tau)d\tau} \int_0^t \frac{\sum_j P_j(\tau)c_{jl}(\tau)\lambda_{jl}(\tau)}{e^{-\int_0^\tau \lambda_l(\eta)d\eta}} d\tau$$

WHERE

$P_l(t)$ = PROBABILITY OF BEING IN STATE l AT TIME t

$\lambda_{jl}(t)$ = TRANSFER RATE FROM STATE j TO STATE l

$\lambda_l(t) = \sum_j \lambda_{lj}(t)$

$c_{jl}(t)$ = COVERAGE ASSOCIATED WITH A FAILURE WHICH, IF COVERAGE WERE PERFECT, WOULD CAUSE A TRANSFER FROM STATE j TO STATE l

THE SYSTEM RELIABILITY IS GIVEN BY

$$R(t) = \sum_{l \epsilon L} P_l(t)$$

FOR THE SET L OF ALLOWABLE STATES.

Figure 13.- CARE III state probability computation.

handling model of figure 10, that information is mapped into time-varying transition rates $\lambda_1(t)$ and $\lambda_2(t)$, as shown in figure 12. What might have been a stationary semi-Markov process for the system of figure 10 will always become a nonstationary Markov process. For large systems, state size reductions of at least 10,000 to 1 have been estimated. The solution to the nonstationary process model of figure 12 is given by the solution to the forward Kolmogorov equation depicted in figure 13. The system reliability is computed by summing the probabilities, $P_\ell(t)$, for the allowable or success states. Numerically, it is more accurate to compute the probability of system failure in lieu of reliability (probability of system survival). The user-defined fault trees specify the system failure states, so that the probability of system failure is simply the sum of $P_\ell(t)$ over ℓ, the set of system failure states. CARE III actually computes the probability of system failure using the equation,

$$Q_\ell(t) = e^{-\int_0^t \lambda_\ell(\tau)d\tau} \int_0^t \frac{\sum_{j\neq\ell} \left[Q_j(\tau) + P_j(\tau)\bar{c}_{j\ell}(\tau)\right]\lambda_{j\ell}(\tau)}{e^{-\int_0^\tau \lambda_\ell(\eta)d\eta}} d\tau$$

where the probability of system failure is given by the sum of $Q_\ell(t)$ over ℓ, the set of system failure states.

3. GLOSS-GATE LOGIC SOFTWARE SIMULATION

It is one thing to implement a very powerful reliability model and quite another to make it useful. For all reliability evaluators, including CARE III, a weakness lies in the unavailability of data for many of the fault-handling parameters. The situation is not a total loss, however, since reasonable engineering estimates can be made in many cases, and furthermore, the sensitivity of the system reliability can be tested against variations in the questionable data. A better way, of course, is to measure or estimate the parameters based on some empirical observations.

3.1 Latent-Fault Modeling and Measurement Methodology

Since system fault detection appears to be the most critical fault-handling parameter, NASA Langley in 1977 initiated a series of studies to investigate a methodology for measuring the fault latency of digital computers (ref. 22). The methodology consisted of simulating a 1000 equivalent gate computer in a host CDC CYBER 173 computer. The simulated computer was a paper design and is referred to as a "hypothetical" machine. The hypothetical machine was simulated at the gate level. Actually, two copies of the hypothetical machine executed identical code in synchronism, where one machine received a stuck-at fault at the onset of the computation. Detection or nondetection was determined after the nonfaulted processor completed its execution. At that time, the computational results of the two simulated machines were compared, bit for bit. Any difference constituted detection. If no detection occurred, the code's input variables were randomly altered, and the processes were repeated for the same fault. This scheme was repeated for up to eight executions for the same fault, if detection did not occur. If a detection occurred in less than or equal to eight repetitions, or no detection occurred after eight repetitions, then a new trial began where another stuck-at fault was induced. This overall process was repeated for up to 1000 randomly selected faults. The 1000 induced faults were selected as a function of piece-part failure rates and were distributed equally across the nodes of the gates. The latency time, i.e., time to fault detection, is expressed in number of code executions or repetitions.

The comparison of output data from two or more computers is often referred to as a comparison-monitoring detector which is an important detection mechanism employed in many operational fault-tolerant systems.

The results of the pilot study were both surprising and intriguing. Using six different programs ranging from a very simple fetch-and-store program to a very complex linear convergence scheme, the pilot study showed that only 50 percent of the induced faults were detected after eight repetitions for all six programs. Figure 14 depicts typical results. The implication that these results have on reliability

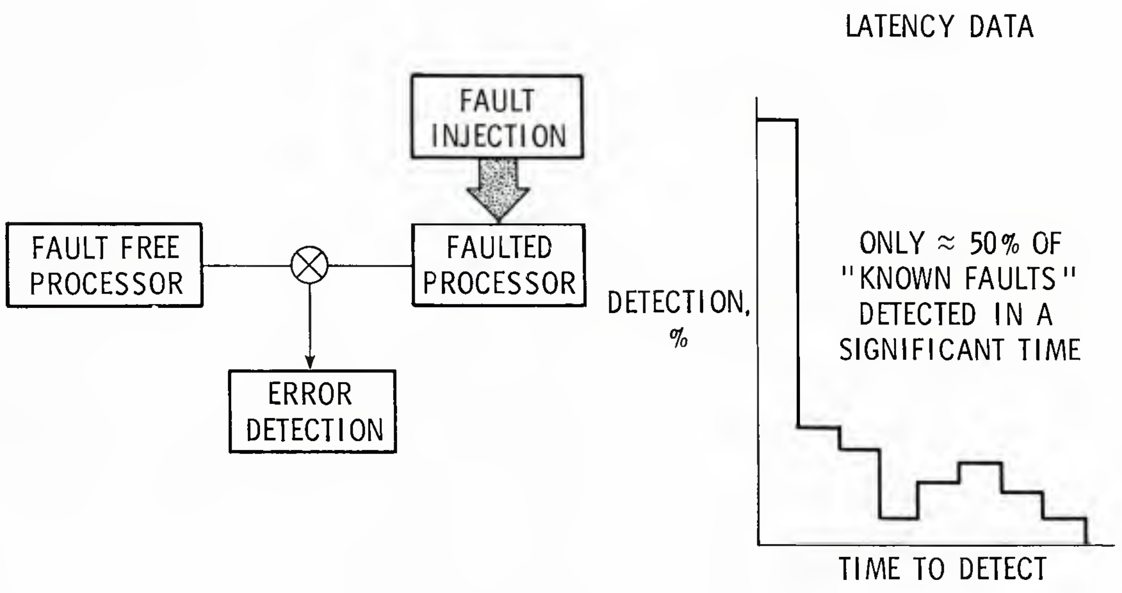


Figure 14.- Latent-fault measurement.

assessment for highly reliable systems is staggering. They suggest that highly reliable systems cannot be designed with comparison monitoring or majority voting as the major stuck-at fault detector (ref. 7).

3.2 Verification of Latent-Fault Measurement Methodology

It was with this concern that a series of further experiments to investigate the validity of the pilot study results were designed at NASA Langley. After all, it was not clear the similar results could be obtained for a real processor executing practical software. The goals of the follow-on work were to test the findings of the pilot study utilizing a real avionic miniprocessor, to access the significance of injecting faults at the gate level and at the functional pin level, to evaluate an airborne self-test program, and to account for undetected faults (refs. 22, 23, 24, and 25). The methodology for gate level simulation, which was codeveloped by NASA Langley and Bendix, is called the GLOSS, Gate Logic Software Simulator.

The pilot study results were tested in three phases using a gate simulation of the Bendix BDX-930 miniprocessor, a 5000 gate equivalent computer. Initially the same six pilot study programs were coded using the comparable primitive instruction set of the hypothetical machine, i.e., load, store, add, subtract, and branch. The next phase allowed the six programs to be recoded using the rich instruction set of the BDX-930, and finally comparison-monitoring detection was measured for a flight control system code in lieu of the six pilot study programs. The surprising outcome of this experiment is typified in figure 15 for all six programs. The percent of undetected faults is about the same for all the programs, instruction sets, and two different machines, i.e., 50 percent. As the code becomes more complex, the shape of the histogram bunches up, so that virtually all the detection occurs in the first execution. The latency time decreases somewhat with increased code complexity, but not the percent detected.

When the same set of experiments are repeated with the exception that faults are induced at the register transfer or pin level in lieu of the gate level, similar results shown by figure 16 appear. One notable difference, however, is that the level of detection significantly rises.

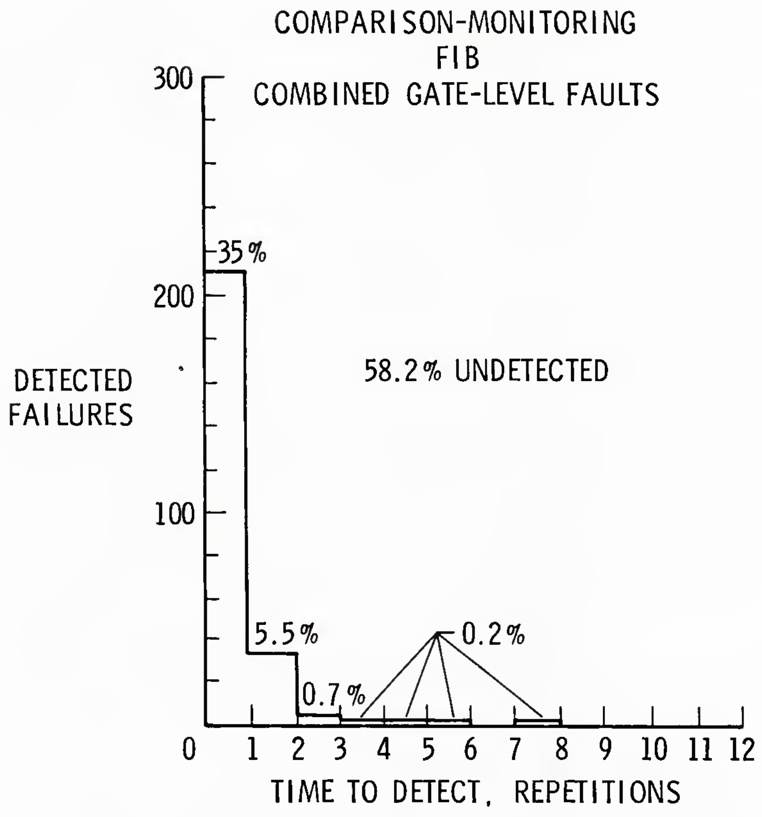


Figure 15.- Fault latency distribution - gate-level faults.

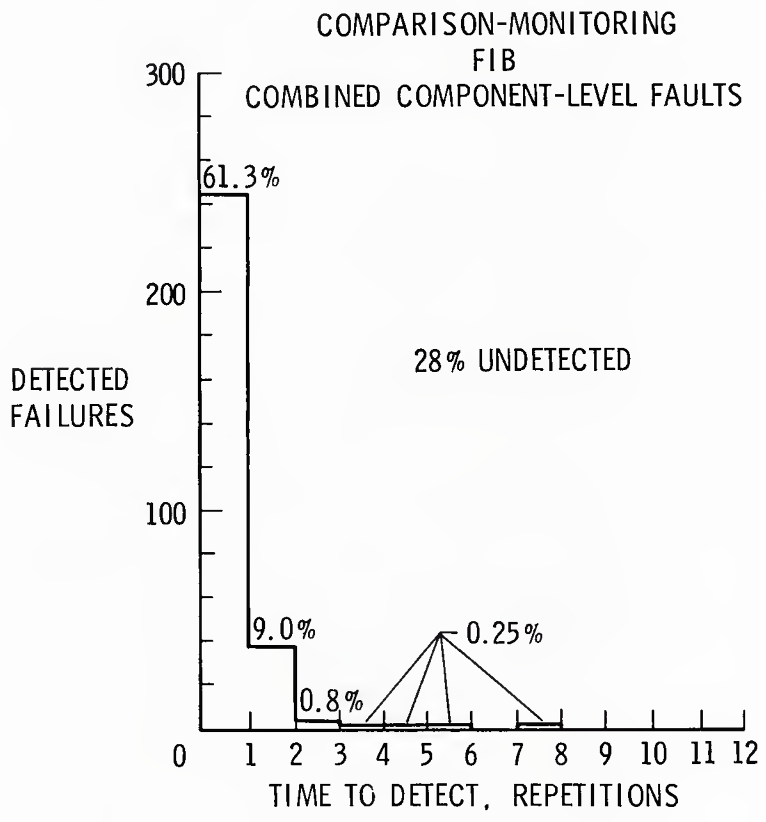


Figure 16.- Fault latency distribution - component-level faults.

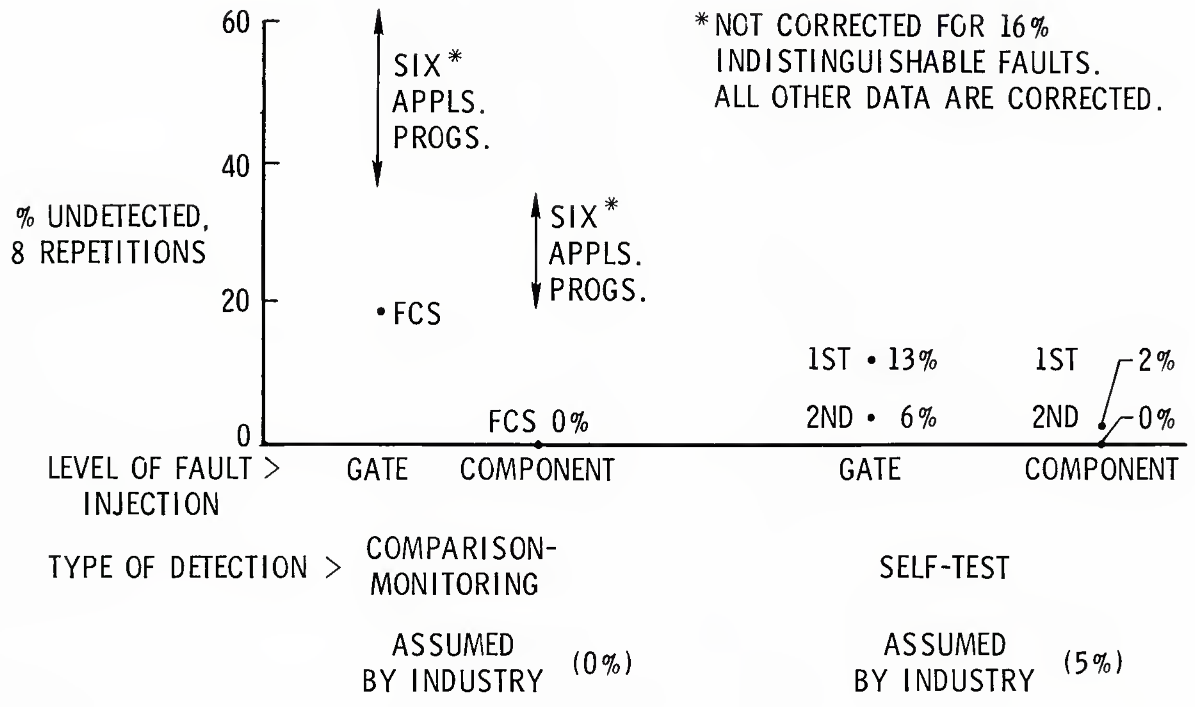


Figure 17.- Summary and results.

As an extension to the pilot study, the latent-fault measurement methodology was applied to an airborne self-test program consisting of 2000 BDX-930 instructions which executed in three milliseconds on the BDX-930. While the simulator executed the self-test program, faults were induced at the gate level and, in a separate experiment, at the pin level. The design goal for the self-test program was 95 percent detection. Figure 17 presents a summary of the self-test detection values and the comparison-monitoring detection values. For the same level of fault inducement, the self-test code shows the highest detection, but fell short of the 95 percent requirement for gate-level faults. With considerable effort and expense, the 87 percent self-test detection was increased to 94 percent, which appears to be a practical upper bound on gate-level fault detection. Flight control system code improved fault detection substantially but still fell short of zero percent undetected for gate-level faults. For component-level injected faults, the industry-assumed value of zero percent undetected was achieved.

3.3 Some Profound Results and Observations

The wide dispersion of detection raises some confounding questions about the method of fault injection and, hence, which detection parameters to use in reliability assessments. The inducement of faults at the gate or pin levels yields a wide dispersion of detection when all other factors are equal. This concern is further exacerbated by the knowledge that the method practiced by industry, pin-level fault injection, yields the higher detection values. At our present level of understanding of fault propagation mechanisms, the pin-level detection values would appear to be nonconservative and should be used with great caution, if at all. This recommendation is based on our knowledge that the gate-level faults that were not detected after eight repetitions are potentially detectable or distinguishable, i.e., there exists some code or sequence of code that will propagate a distinguishable fault.

In the process of investigating the reason why faults were not detected after eight repetitions, it was discovered that there exists a class of faults that can never have an effect on the system and, therefore, will never be detected. This class of indistinguishable faults has been estimated to comprise 16 percent of all faults. An example of an indistinguishable fault is a stuck-at fault located at the unused output of a flip-flop circuit. An important outcome of this discovery regards the method of estimating detection coverage. The more accurate approach is to delete the 16 percent indistinguishables from the set of induced faults in the computation of detection coverage. The net effect is to increase the magnitude of detection coverage.

The lessons learned from these latent-fault modeling and measurement studies are summarized as follows:

- Practical measurement of detection coverage for stuck-at faults is possible and is a necessary aspect of reliability assessment.
- Comparison-monitoring detection for typical application codes is much less than expected, which poses serious implications for highly reliable systems.
- 95 percent gate-level self-test detection coverage is measurable and achievable, but expensive to accomplish.
- The industry practice of measuring self-test detection by inducing faults at the pin level may not be conservative, and in view of the fact that the reliability of highly reliable systems is very sensitive to detection, further analysis of this practice is required.

4. CONCLUDING REMARKS

The CARE III program and the user-friendly interface, CARE3MENU, are currently available to U.S. companies through NASA's software dissemination facility, COSMIC (ref 26). The CARE III math model is embodied in a Fortran 77 computer program that has received considerable national scrutiny. The validation of CARE III was conducted by industry, the university community, and by the U.S. Government at NASA Langley Research Center and by the U.S. Air Force at Wright-Patterson Air Force Base.

The development of a generally applicable GLOSS computer program, which embodies the GLOSS methodology, is currently underway. The GLOSS will execute on the VAX-11, 700 computers series (ref. 27).

REFERENCES

1. Mathur, F. P.: Reliability Study of Fault-Tolerant Computers in Supporting Research and Advanced Development. Jet Propulsion Laboratory, Aug. 1969, Space Programs Summary 37-58, Vol. III, pp. 106-113.

2. Blazek, R. H. et al.: Demonstration of Combined Reliability Prediction and Verification Techniques to a Typical Flight Control System, Vol. I, Development and Application of Tabular System Reliability Analysis to the F-111 Pitch Flight Control System. Battelle Columbus Laboratories, AFFDLOTR-71-128, Vol. I. (Available from AF Flight Dynamics Lab, Wright-Patterson AFB), Oct. 1971.

3. Roth, J. P. et al.: Phase II of an Architectural Study for a Self-Repairing Computer. SAMSO TR-67-106, U.S. Air Force, Nov. 1967. (Available from DTIC as AD 825460.)

4. Bouricius, W. G. et al.: Reliability Modeling Techniques and Trade-Off Studies for Self-Repairing Computers. RC 2378, Res-Div., IBM Corp., Feb. 1969.

5. Raytheon Company, Sudbury, MA: Reliability Model Derivation of a Fault-Tolerant, Dual, Spare-Switching Digital Computer System. NASACR-132441, 1974.

6. Raytheon Company, Sudbury, MA: An Engineering Treatise on the CARE II Dual Mode and Coverage Models. NASA CR-144993, 1976.

7. Bavuso, S. J.: Impact of Coverage on the Reliability of a Fault Tolerant Computer. NASA TN D-7938, 1975.

8. Ultra-Systems, Inc., Newport Beach, CA: Reconfigurable Computer Systems Study. NASA CR-132537, 1974.

9. Bjurman, B. E. et al.: Airborne Advanced Reconfigurable Computer System (ARCS). The Boeing Commercial Airplane Company. NASA CR-145024, 1976.

10. O'Neill, E. J.; and Halverson, J. R.: Study of Intermittent Field Hardware Failure Data in Digital Electronics. Sperry Univac Defense Systems, St. Paul, MN. NASA CR-159268, 1980.

11. Masson, G. M.: Executive Summary - Intermittent/Transient Faults in Computer Systems. The Johns Hopkins University. NASA CR-159229, 1979.

12. Nagel, P. M.: Software Reliability: Repetitive Run Experimentation and Modeling. Boeing Computer Services Company. NASA CR-165836, 1982.

13. Stiffler, J. J. et al.: CARE III Final Report Phase 1, Vols. 1 and 2. Raytheon Co., Sudbury, MA. NASA CR-159122 and NASA CR-159123, 1979.

14. Bavuso, S. J.: Trends in Reliability Modeling Technology for Fault Tolerant Systems. AGARD Conf. Proc. No. 261 on Avionics Reliability, Its Techniques and Related Disciplines, April 1979.

15. Stiffler, J. J.; and Bryant, L. A.: CRE III Phase II Report, Mathematical Description. Raytheon Co., Sudbury, MA. NASA CR-3566, 1982.

16. Hopkins, A. L.; and Smith, T. B.: The Architectural Elements of a Symmetric Fault-Tolerant Multiprocessor. IEEE Trans. on Computers, Vol. C-24, No. 5, 1975.

17. Osder, S.: The DC-9-80 Digital Flight Guidance System's Monitoring Techniques. Sperry Flight Systems, Phoenix, AZ, AIAA Paper 79-1704, 1979.

18. O'Hern, E. A.: Space Shuttle Avionics Redundancy Management. Rockwell International, AIAA Digital Avionics Systems Conference, April 1975.

19. Wensley, J. H. et al.: Design Study of Software-Implemented Fault Tolerance (SIFT) Computer, SRI International, Menlo Park, CA. NASA CR-3011, 1978.

20. Conrad, C. L. et al.: CARE III User Friendly Interface User's Guide. Research Triangle Institute, Research Triangle Park, NC. NASA CR-172608, 1985.

21. Stiffler, J. J.: Fault Coverage and the Point of Diminishing Returns. Journal of Design Automation and Fault Tolerant Computing, Vol. 2, No. 4, Oct. 1978.

22. Trivedi, K. S.; and Geist, R. M.: A Tutorial on the CARE III Approach to Reliability Modeling. Duke University, Durham, NC. NASA CR-3488, 1981.

23. Nagel, P. M.: Modeling of a Latent Fault Detector in a Digital System. Vought Corp., Hampton, VA. NASA CR-145371, 1978.

24. McGough, J. G.; and Swern, F. L.: Measurement of Fault Latency in a Digital Avionic Miniprocessor. Bendix Corp., Teterboro, NJ. NASA CR-3462, 1981.

25. McGough, J. G. et al.: Methodology for Measurement of Fault Latency in a Digital Avionic Miniprocessor, AGARD Cong. Proc. No. 303 on Tactical Airborne Distributed Computing and Networks, June 1981.

26. Bavuso, S. J. et al.: Latent Fault Modeling and Measurement Methodology for Application to Digital Flight Controls. Advanced Flight Control Symposium, USAF Academy, Colorado Springs, CO, Aug. 1981.

27. Bavuso, S. J. et al.: CARE III Model Overview and User's Guide (first revision). NASA TM 86404, 1985.

28. McGough, J. G. et al.: Feasibility Study for a Generalized Gate Logic Software Simulator. NASA CR-172159, 1983.

I S A U R E

Integration of Security and fAUlt tolerance

in the REalite 2000 system

J.CAZIN (*) and P. MICHEL (*)

1. INTRODUCTION

In the framework of the french project **SURF** sponsored by ADI (**), the ONERA/CERT research center (*) and the INTERTECHNIQUE Company (***) have developped a fault tolerant operating system, named **ISAURE.**

This system includes data protection features and error recovery mechanisms. It offers the same services as the REALITE 2000 system already available on the market. It used first the MULTI6-INTERTECHNIQUE microprogrammable machine, and later a specific hardware implementing protection domains via capability based adressing mechanism.

The collaboration between the two organisms is the result of converging and complementary aims :

. the INTERTECHNIQUE Company is trying to set up a future range of systems and machines oriented towards management applications and offering a high level of functional security ;

. the CERT/ONERA research center wishes to apply methodological research results /1, 2/ and thus continues its research work in the context of practical applications.

This collaboration is one of the more original features of the project and it allows the introduction of advanced concepts into the industrial sector.

1.1. Services offered by the REALITE 2000 system

REALITE 2000 is a management oriented system. It functions in a real time conversational context and is available to multiple users (up to 32). It manages a data base by using a virtual memory system.

Users dialogue with the system through display devices. All communications are made through either easily assimilable languages adapted to the applications of the system : JCL, Text editor, BASIC-management, FRANCAIS (interrogative language very close to the usual French thus allowing the data base to be consulted easily) or preestablished specific application programs.

The data base is decomposed in a four levels file hierarchy :

system dictionary --> user dictionary --> file dictionary --> data file

The file structure allows the manipulation of items of variable lengths and also directs access to them, thus ensures a compromise between performance and storage efficiency.

REALITE 2000 system, by using a virtual memory technique, offers the user's programs a memory capacity equal to that of the mass storage connected on line to the system.

1.2. Additional user needs

In the context of business management, it is extremely important that the management tools be infallible, as any defect may cause perturbations which could be fatal for the firms using these tools. We may observe several critical aspects :

. security and coherence of information in the data base, which constitutes the user's most important holdings ;

. availability and continous functioning of the system even when incidents are present.

They are several causes for incidents (or exceptions) :

. user error of malice ;

. misuse encouraged by the system itself ;

. software error in either the operating system or the application programs ;

. hardware failures.

(*) ONERA-CERT : Centre d'Etudes et de Recherches de Toulouse - 2, av. Edouard Belin BP 4025 - 31055 TOULOUSE CEDEX - FRANCE

(**) ADI : Agence de l'Informatique - Tour Fiat Cedex 16 92084 PARIS LA DEFENSE - FRANCE

(***) INTERTECHNIQUE : Société Intertechnique - B.P. 1 - ZI "Les gatines" 78370 PLAISIR - FRANCE

1.3. General specifications for ISAURE

The general remarks in 1.2. reinforced by the conclusions of a critical study of REALITE 2000 are at the origins of the basic specifications of the ISAURE system :

. make use of the services of the REALITE 2000 system ;

. offer a safer policy for using the system (strict dynamic management of privileges and rights, a more orderly use of the data base, more elaborate tools for programming applications) ;

. set up a very reliable structure and organization of the system, based on firmly established principles : design methodologies /3,4/, error confinement and data protection /5,6,7,8,9/ ;

. offer error recovery mechanisms based on exception handling principles /10,11,12,13,14/ to deal with incidents mentionned in 1.2.

These basic specifications have led to important technical choices :

. develop a PASCAL-like programming language adapted to modular systems (this choice, moreover has eliminated many problems inherent to programming a system in an assembly language) ;

. emulate by microprogram a capability-based adressing machine and later built up a specific hardware.

In the two following sections, we set up the problems of error confinement and error recovery and present the solutions (formalism and tools) choosen in the ISAURE system.

The last section deals with a detailed example : error recovery in the ISAURE data base management.

2. ERROR CONFINEMENT IN A MODULAR AND HIERARCHICAL SYSTEM

2.1 Error confinement and hierarchy

The design phase of a program (or a system), by stepwise refinement, leads to a hierarchy and to a decomposition of the program into types or functions. According to their level in the hierarchy and the chosen formalism, they are implemented by modules, monitors, clusters, procedures, blocks, instructions... /3,4,15,16/. We will use, later on, the word **component** to name one of these implementation units.

Considering program structures, two kinds of ordering relations between components within a hierarchy must be pointed out :

. **the "nesting relation"** ; for example, a procedure nests a block, a block nests another block (in ALGOL-like langages).
The nested component context, i.e. the set of reachable data from the component, is the union of local data of the component and of the nesting component context.

. **the "calling relation"** ; for example, a module calls another module, a procedure calls another procedure.
The contexts of calling and called component are dissociated. A mechanism for parameter passing is needed to transfer some information between these two components.

The nesting and calling relations are generally used together. In an ALGOL program, for example, the nesting relation is prevailing with nested blocks but there are also procedure calls. In a domain system, the relation between two domains is always the calling relation but a domain nests procedures.

The distinction between these two kinds of relation is important for error confinement : data isolation uses the calling relation. Intersection between the contexts of two components related in the calling relation is reduced to the necessary minimum, i.e. the set of parameters between these components.

Therefore, the error confinement principle implies not to use the nesting relation at least before some level of refinement. Error confinement exists between components generated at this level. This approach is found in systems structured in domains, like ISAURE, where error confinement exists between domains but does not exist between lower level components inside a module (procedures, blocks).

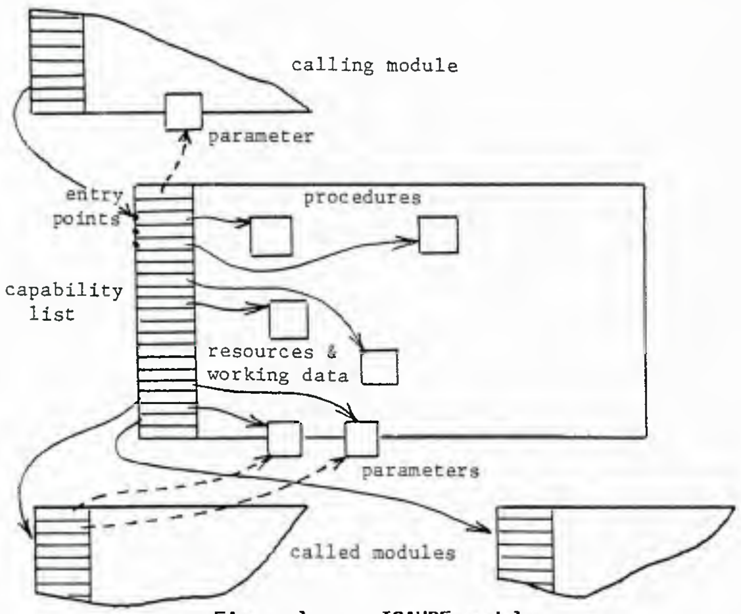
Some data have a life-time longer than the execution time of the components that use them (for example a whole session time for book-keeping data, or several days for files). Partitionned systems generally use the notion of "remanent data" : the value of a remanent data is preserved from one execution of the component which contains it, to the next one.

2.2. Modularity and hierarchy in the ISAURE system

The ISAURE system uses the error confinement principles as defined above. Its basic component is the module and the nesting relation appears only at refinement levels below module level.

ISAURE is a multi-process system. A process runs in a hierarchy of modules. The run time context of a process, i.e. the set of objets reachable by the process, changes everytime a module is entered. Parameter passing is the only way modules can communicate. A parameter object belongs to a calling module and can be accessed from a called one as long as the process is running this module.

A module is a set of closely related objects (resource, access functions, working data...). The ISAURE system runs on a capability-based addressing computer ; every access to an object is made through and under control of a capability (type control, object overflow control, right control). Then a list of capabilities represents a module. Basic objects are segments like data segments or program segments, or capabilities segments (capability-list or C-list).

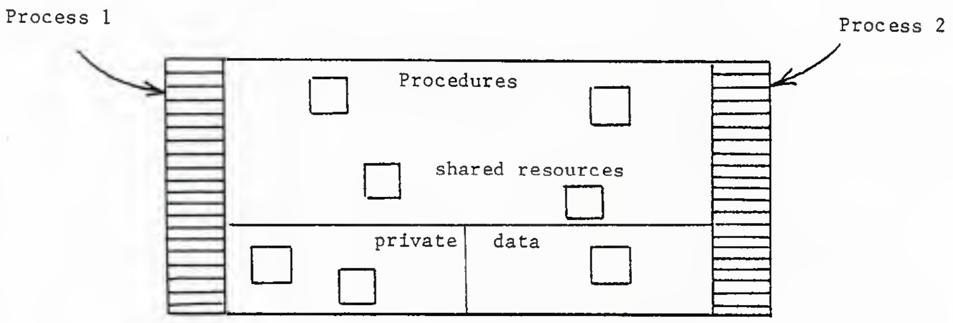


- Figure 1 : an ISAURE module -

Inter-process communications and resource sharing are made in particular modules : the monitors. So three types of modules are defined in the ISAURE system :

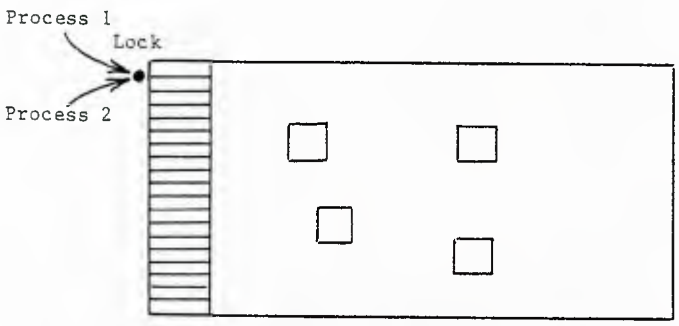
. **private modules or domains** : they consist of a set of private objects accessed through a private C-list ;

. **general monitors or G-monitors** : they allow inter-processes communication and resource sharing. A G-monitor is re-entrant and consists of a set of shared functions (procedures), one or more shared resources and a set of private data allowing re-entrance (i.e. one set of data per process). Each hierarchy owns an actual occurence of such a G-monitor, that is a private C-list ;



- Figure 2 : a G-monitor -

. **simple monitors** : they are shared, but not re-entrant. Their implementation is far easier than for G-monitors. Not private data are needed in such a module and every object is shared : neither re-entrance nor synchronization is to be ensured. Such a monitor is represented by a shared C-list, and all processes may access it. This kind of module turned out to be efficient for basic resource management (virtual memory overflow management, for example) or for specific processing (error messages management).



- Figure 3 : a non re-entrant monitor -

Every module in the ISAURE system is described by an interface and a body.

The interface describes the external aspect of the module (entry points, parameters) for calling modules. The body describes the set of internal objects of the module (data, procedure) and the set of called modules.

The interface and the body description are both divided into a normal and an exception area ; the last one is described in the next section.

```
INTERFACE file ;
   ENTRY readitem :
     key : keyitem (R) ; % read only access (keyitem = predefined type)%
     buffer : item (W) ; % write only access (item = predefined type)%
END
   ENTRY writeitem :
     key : keyitem (R) ;
     buffer : item (R) ;
     END
EXCEPTION AREA ;
     <exception area>
END                  %interface%

PATTERN file ;
   SHORT DATA (R,W)                              %example of data%
     v1, v2 : TALLY ;
     d      : DOUBLE ;
END
     -
     -
     -
     Ptfile : CAPABILITY (R,W,C) ;              % pointer of file%
     -
     -
     -
     p1     : IPROCEDURE ;                       % internal procedure%
     -
     -
     -
     CALLED mem-alloc (getframe, releaseframe) ; % called module%
EXCEPTION AREA ;
     <exception area>
END                 % pattern %
```

- Figure 4 : an example of ISAURE system -

3. ERROR RECOVERY IN MODULAR SYSTEMS

Several steps can be distinguished in fault treatment :

. error detection ;

. error localization that is determinating what objects caused the fault, and evaluating damage ;

. error recovery consisting of :
  * damage repair, that is putting the system back into a coherent state ;
  * restarting execution from this coherent state.

This state is either the current state (forward error recovery) or a previous coherent state (backward error recovery).

The above defined functions are called curative processing ; sometimes they imply preliminary processing (that we shall name "**monitoring functions**") such as safeguard in a backward error recovery strategy. We shall name "**exception handling**" the set of all these functions and "**exceptions**", the events which start them : error detection or monitoring function call.

3.1. Exception handling in modular systems

Exception handling in hierarchical and modular system is based on the principle of error masking :

. we try to recover an error in one component, i.e. to mask the error for the upper level components ;

. if this recovery cannot be done, the error is propagated toward the immediate upper component where error masking will be tried again.

The important problem which is set up is to know at which level error masking is possible ; we shall name this level the **recovery level.** Error recovery cannot be done in one component either when error localization cannot be done inside this component or when error localization shows that upper levels are involved in the error. Therefore, determining the recovery level depends on error localization : the recovery level is the highest level involved in the error if localization has precisely determinated this level or an adequate level which ensures recovery by an over-estimation of the error effects.

Viewed from a component, a detection of exception is :

. an internal detection, made by an explicit detector or **assertion** ;

. or an implicit detection coming from a lower-level component (hardware or software) and signaled by an exception raising mechanism.

When an exception is detected, it must be taken into account even if its causes are unknown ; every exception must always be associated with a handler. Dynamic association between exceptions and handlers is interesting to provide flexibility in programs.

Exception handling consists, as formely said, in restoring data in a consistent state and in restarting the normal processing. Consistency generally concerns data at several levels. A handler located in a module cannot correct erroneous data in lower levels, and must therefore call some recovery functions at these levels ; we shall name **sub-handlers** the functions recovering residual informations.

Restarting normal processing corresponds to the end of an exception handler on the recovery level ; two kinds of restart must be considered :

. a return to the running point, i.e. after the operation which raised the exception ; it corresponds to a forward error recovery and we shall call it a **CONTINUE** restart ;

. a new execution of the operation in which the exception was raised ; it corresponds to a backward error recovery and we shall call it a **RETRY**.

If the current level is not the recovery level, a handler must raise an exception for a higher level ; we shall call this case an **ABANDON**.

3.2. Tools for exception handling in ISAURE

We have so far used in this section the general term "component" without saying anything about its nature : module, procedure, block... A module is the basic component in the ISAURE system : declarations of exceptions and handlers are made as any other declaration at the module level. Nevertheless, some exceptions deal with components smaller than a module (e.g. a procedure or an instruction) and recovery must be provided at their level. In order to solve this problem a recovery level may be associated to an exception inside a module ; we shall see that the recovery level associated to an exception will be induced by the location of the association between the exception and a handler.

We are now going to present the choices that have been made in the ISAURE system, beginning with linguistics tools. As we have seen in the previous section, an ISAURE module is divided into two areas : a normal area and an exception area, which brings modularity and independence between normal and exceptional treatments. Declarations and associations presented below will be located in the exception area of the modules.

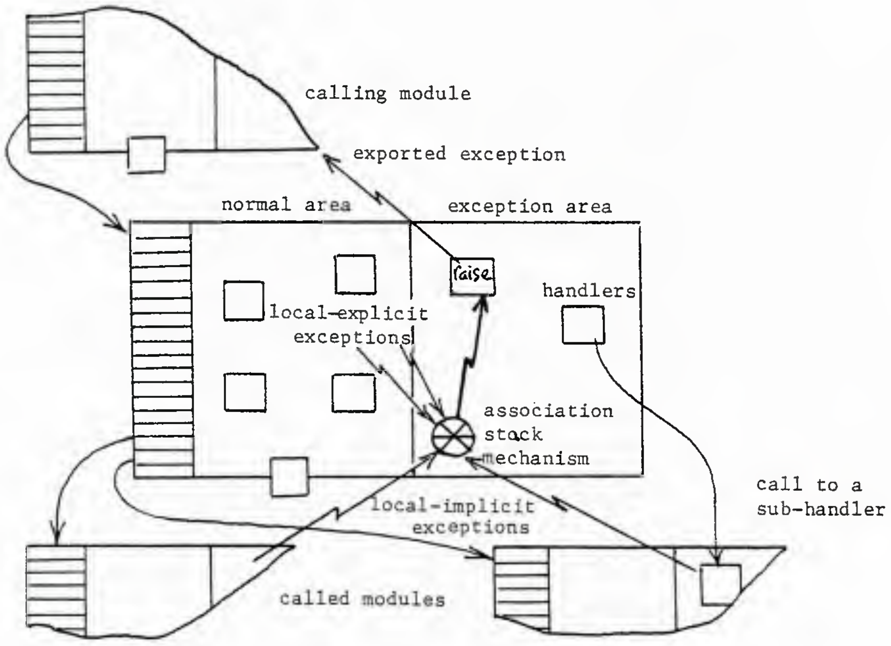


Figure 5 : Exception handling in a module

3.2.1. Declaration of exceptions

We can find two kinds of exceptions in a module :

. **local exceptions.** They are either explicit and declared in the exception area of the module or implicit, i.e. defined and raised by lower level modules ;

. **exported exceptions.** They are explicit and declared in the interface of the module in order to be reported to the calling modules.

Let us point that an exported exception of a module is considered as a local and implicit exception by a calling module. Exceptions raised by hardware are a particular case of implicit exceptions in a software module and are predefined in the language.

3.2.2. Declaration of exception handlers and sub-handlers

An exception handler is a particular procedure in the exception area of a module. A sub-handler is a special entry point, defined in the exception area of the interface of a module (see figure 6).

The translator verifies that a sub-handler is only called from the exception area of a calling module, i.e. by an exception handler. Let us point that calling a normal entry point of a module from an exception handler of a higher module is allowed and may even be very useful if we want, for example, to undo by a reverse function what has been done by an entry point, the reverse function being an other entry point.

3.2.3. Definition of associations between exceptions and handlers

Associations in a module only concern local (either explicit or implicit) exceptions of the module. An association between an exception and a handler defines the handler to start on an occurence of the exception ; it may (optionally) also defines the restart mode (CONTINUE or RETRY) at the end of exception handling.

Syntax [<exception-name> --> <handler-name> {(<restart-mode>)}]

Example [errformat --> salvation]

```
        [b --> backup (RETRY)]
```

Two kinds of assocations are possible :

. **Statics associations**

A static association is valid during the whole execution in the module. It may be an **explicit association** defined in the exception area of the module.

For an implicit exception, i.e. an exception raised from a lower module, it may be an **implicit association** ; this kind of association and the handlers associated are predefined and stored in a library. So implicit handlers and associations are predefined for exceptions raised from hardware or basic parts of the operating system so as to avoid the programmer to always take them into account.

An implicit association prevails over an implicit association.

. **Dynamics associations**

They are explicitely defined inside the program. The scope of a dynamic association is a component (**procedure, block, or instruction**) ; this component depends on the location of the association definition :

. it is the called procedure for an association defined after a procedure call instruction ;

. it is a block for an association defined after a "begin" keyword ;

. otherwise, it is the instruction which comes before the association.

Assocations concerning one exception are recorded on a stack so that the association of the finest level may prevail : an association on the instruction level prevails over an association on the block level which prevails over an association on the procedure level which prevails on a static association (module level).

3.2.4. Occurence exception

An implicit exception is raised from a lower level module. It is an implicit occurence of exception for the current module.

An explicit exception is explicitely raised inside the current module by an assertion test.

Syntax IF <condition> RAISE <exception-name>

If the exception is a local exception, its occurence will lead to the execution of the associated local handler. If it is an exported exception, its occurence will be transmitted to the calling module.

Remark : An occurence of an exception (local exception) for which there is no association, activates a standard default handler (the same for every exception) which sends a message to the user and raises an exported exception (i.e. ends by an ABANDON).

3.2.5. End of exception handling

The execution of an exception handler ends on one of the following instructions :

. **RAISE** <exported-exception> : the handler raise an exception for the calling module ;

. **RETRY** : the execution of the component interrupted by the exception occurence is retried. The restarted component (module, procedure, block or instruction) is given by the association stack formerly mentionned ;

. **CONTINUE** : the execution restart at the end of the interrupted component ;

. **RETURN** : the restart mode (RETRY or CONTINUE) is given by the association used on the exception occurence.

3.2.6. Hardware and microprogrammed features

We consider only tools that concern fault tolerance and just list them without any details.

We can distinguish two categories :

. **features for error detection and error confinent :**
* classical hardware controls : parity check, transfer check,
* capability-based addressing with type, access right and overflow controls,
* instructions for domain commutation : ENTERDOM instruction with parameter passing and RETURNDOM instruction.

. **features for exception management :**
* instructions for software exception raising : RAISE instruction for any internal exception starts the execution of a local handler ; RETURDOM instruction for an exported exception reports the exception up to the calling module ;
* handler starting mechanism : it takes into account the exception raising and uses an execution transfer stack which is updated by associations ;
* special return instructions : they implement the handler ends (RETRY or CONTINUE modes).

3.2.7. Software features

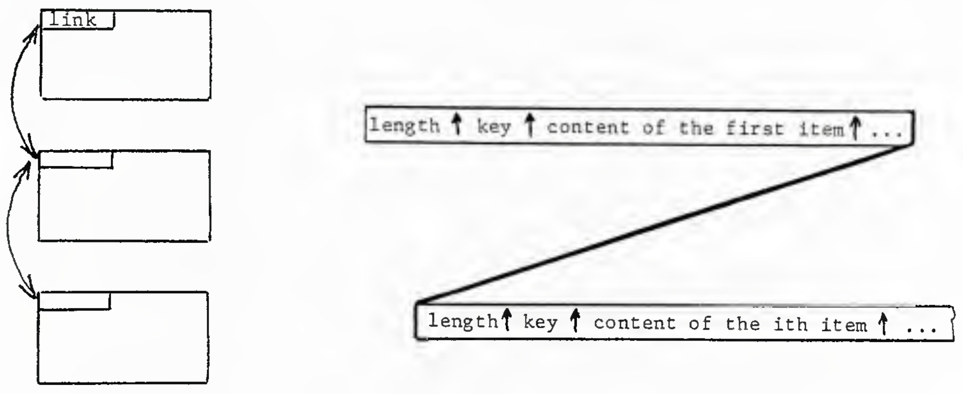
Software features for exception handling are distributed throughout the whole ISAURE operating system. In the next section we will just present errors recovery in the ISAURE data base.

4. ERROR RECOVERY IN THE ISAURE DATA BASE

4.1. Organization of the ISAURE file management

A file is composed of one or several "groups". A group is a set of linked pages of virtual memory (we shall call them "frames" according to the constructor terminology) ; it contains a sequence of items. Items are delimited by a special separator byte. The first field of an item gives the length of the item, the second field is the key of the item.

So, a file is composed of two types of informations : structure description (links for frames, separators and length of items) and user data.


link
length ↑ key ↑ content of the first item ↑ ...
length ↑ key ↑ content of the ith item ↑ ...

A group = A set of linked frames

A group = A sequence of items

Figure 6 : file structure

In the ISAURE operating system each active file is encapsulated with its acces functions in a module (a private module if the file is not shared or a G-monitor in the other case ; cf. §2). The module are integrated in the following hierarchical structure.

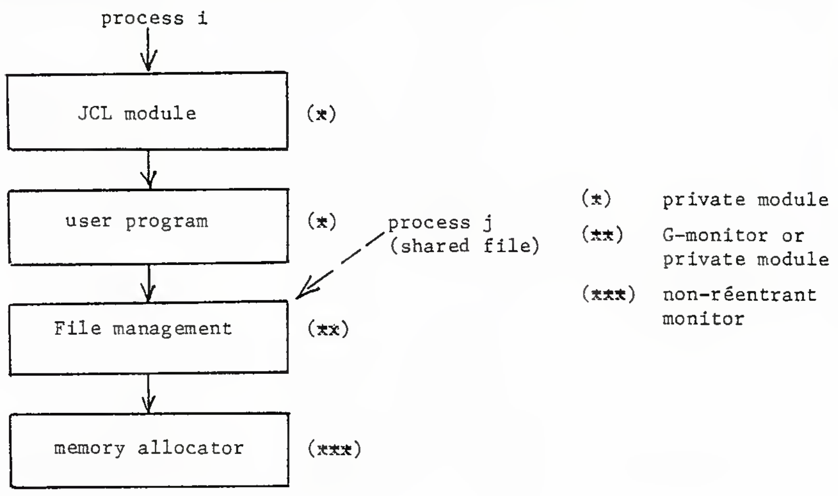


Figure 7

In the context of business applications, a user program deals with cyclic or repetitive executions. We shall call **transactions** these executions. A transaction uses the services of a file management module for access operations on a file. A file management modules uses the services of the memory allocator monitor to get and release virtual memory frames.

4.2. Objectives of the exception handling

An incident may have various origins (user mistake, software or hardware failure) and may cause a loss or an alteration of information in the data base. The damage may be limited to structure information or may be spred also to user data. In the first case detection, exception raising and exception handling are realized in the file module where structure organization is known whereas in the second case they must be located in the user module for a more global restoration.

The objective being to recover the various informations (partially or entirely),several recovery strategies are proposed : recovery of the structure with a SALVATION HANDLER or entire restoration of all information with a BACKUP HANDLER /17/.

In the following paragraphs, with previous formalism and tools, we present the scheme of the general strategy and we detail the principles of the two programs ; first we consider a context of non interactive processes and then we look at the case of shared files.

4.3. General scheme of the strategy

The strategy of data base recovery concerns essentially the user program and file management modules.

In normal running, the user module calls periodically a safeguard handler by raising a monitoring exception associated with it to prevent a data base error. Its role is to record a copy of modified data.

During the execution of its operations, the file management module controls and tests various assertions ; when a test is negative, an exception is raised and the associated handler is called ; two cases must be considered :

. local exception ; the associated local handler is named SALVATION HANDLER ;

. exported exception with propagation to the user module ; the associated handler in this last module is named BACKUP HANDLER.

Note that the SALVATION HANDLER can restore only a partial coherent structure in this case ; the user module may call the BACKUP HANDLER for a total restoration.

In all these strategies the executions of handlers need the call of the virtual memory allocator monitor to release residual frames and to ensure a correct state of memory. The figure 9 gives an illustration of this discussion and figure 8 presents an example of the programming formalism.

```
INTERFACE user ;
   ENTRY transaction ;
END

PATTERN user ;
   key : keyitem (R,W,P) ;
   buf : item (R,W,P) ;
   ...
   operate : IPROCEDURE ;                       % internal procedure %
   CALLED file (readitem, writeitem) ;
EXCEPTION AREA ;
   s,b : EXCEPTION ;
   herrfile : HANDLER ;
  safeguard : HANDLER ;
  backup : HANDLER ;
  ...
  [errfile --> herrfile] ;                     % statics assocations %
  [s --> safeguard (CONTINUE)] ;
  [b --> back up (RETRY)] ;
END

PROCEDURE transaction ;
   BEGIN
      % initial processing %
   ...
      test := TRUE ;
      WHILE (test) DO
         BEGIN
         RAISE s ;                            % explicit occurence
         ENTER file. readitem (key,buf) ;     % module call %
         operate ; [errfile --> backup (RETRY)] %procedure call* assoc %
         ENTER file. writeitem (key, buf) ;
         END
RETURNDOM ;
    END

INTERFACE file ;
   ENTRY readitem : key : keyitem (R) ; buffer : item (W) ; END
   ENTRY writeitem : key : keyitem (R) ; buffer : item (R) ; END
EXCEPTION AREA ;
   errfile : EXCEPTION ;
END

PATTERN file ;
   % declarations of normal data and procedures (cf. figure 4) %
   ...
   CALLED mem-alloc (getframe, releaseframe)   % called module %
EXCEPTION AREA ;
   errformat : EXCEPTION ;                     % local exception %
   salvation : HANDLER ;
   ...
   [errformat --> salvation] ;                 % static association %
END
PROCEDURE readitem ;                            % description of procedures %
```

Figure 8 : user and file modules

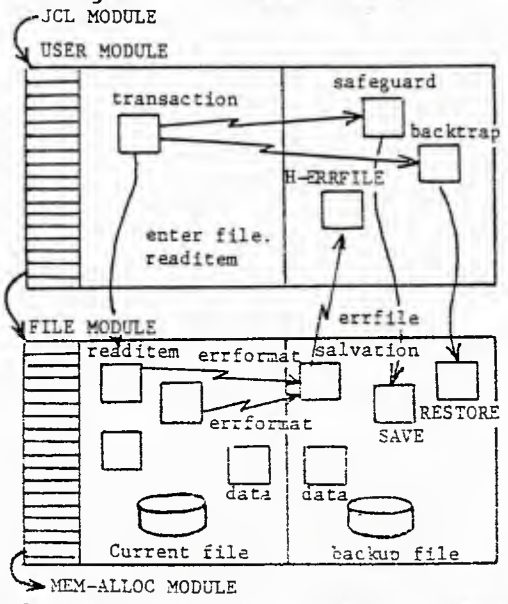


Figure 9 : Recovery strategy in ISAURE file management

Recovery of information structure : the salvation handler

Its objective is to set back up, when possible, the links between frames or the length of items from redundant informations (forward and backward links between frames, lengths and separators of items). This handling does not ensures that the file will be entirely repaired ; for example it doesn't repair user data damages.

Total recovery of the file : the backup handling

Its objective is to allow the recovery of all user's and structure informations in a consistent states. To this end, two handlers have been implemented :

. a preventive handler (the safeguard handler) which starts during normal processing. It establishes a checkpoint by recording a copy (a "backup version") of the current, presumably correct, state of the file. This recording is made after a fixed number (which may be modulated) of modifications in the file ;

. a currative handler (the backup handler) which starts when a crash is detected. It restores the file in a correct state, i.e. the state of the previous checkpoint.

4.4. Case of interactive processes sharing one or several files

The recovery of interactive processes sharing files leads to the problem of coherent restoration and restart for all process /18,19/.

For that purpose, we use the notion of **transactions** for the synchronization of the safeguard of these shared files : a safeguard may occur when each process is at the beginning of a transaction. Thus, each beginning of transaction is considered, for a process, as a possible recovery point (called an **aptitude point**) and raises an exception (the "aptitude exception").

The safeguard is automatically invoked each N transactions (this frequency is a parameter of the application) when the current process starts the transaction. The safeguard handler will be actually run when all the other processes will have raised the aptitude exception, safeguard and aptitude handlers having then locked all the processes.

Figure 10 gives an illustration of the above discussion.

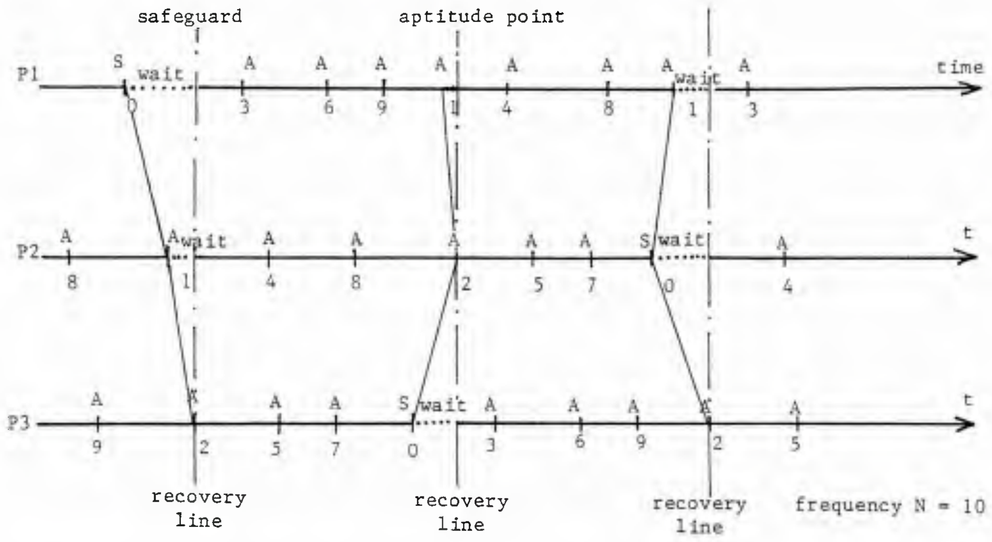


Figure 10

When an incident occurs in the system or in a USER module, recovery consists in activating the backup handler which restores the state of the last checkpoint of the file and which informs users of the transaction number from where to restart.

To implement this strategy we have integrated monitoring exceptions of aptitude inside USER modules and synchronization manager inside FILE modules.

Coherence for several files

We will discuss this problem with the example of two processes P1 and P2 which share the file F2 and which own a private file, respectively F1 and F3 (see figure 11) :

Suppose that P1 raises a safeguard exception : file F1 can be saved but in the case of F2 we must wait for an authorization of P2 (aptitude exception). In this context, to maintain coherence if an incident appears, it is necessary to ensure the saveguard of the three files at the same time. So a checkpoint concerns a set of files and not only one file and is effectively done only when the aptitude exception has been raised by all the processes using at least one of these files.

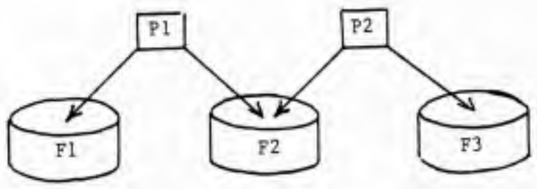


Figure 11

5. CONCLUSION

Modular and hierarchical conception of the ISAURE system has allowed us to carry out the error confinement and recovery principles and to develop appropriated techniques and tools.

The error recovery functions in the files which were presented in the last section of this paper, show an example of using these tools.

These principles and tools have been used in specification and implementation of every function in the ISAURE system : virtual memory management, overflow management, directories or commands management.

We can say to conclude that the interest of the ISAURE system stays in using a set of techniques and in trying to define the best association of them for the system reliability : stepwise refinement conception method, modularity and exception handling features in the programming language, error recovery techniques in data base, error confinement features, capability-based addressing machine.

ACKNOWLEGMENTS

We would like to thank the others participants of the ISAURE project for their useful and friendly contribution during four years : L. BOI from ONERA/CERT, M. BUIS, M. BAUMGARTEN, J.M. JANTKE from INTERTECHNIQUE Company and A. PLAS from UPS university of Toulouse.

REFERENCES

/1/ L.BOI, P.MICHEL
"Design and principles of a fault tolerant system"
Proceedings of 3rd Conference on Software-engineering
Atlanta, (May 1978)

/2/ L.BOI, P.MICHEL, M.BUIS, J.M. JANTKE, J.CAZIN, A.PLAS
"ISAURE : a modular system for data security and fault tolerance"
IFIP Working Conference on "Reliable Computing and Fault Tolerance in the 1980's", London, (Sept. 1979)

/3/ B.H. LISKOV
"A design methodology for reliable software systems"
Proc. of AFIPS-FJCC, vol. 41 (dec. 72)

/4/ D.L. PARNAS
"The influence of software structure on reliability"
Conf. on reliable soft. proc. ACM (April 75)

/5/ R.S.FABRY
"Capability-based addressing"
Univ. of California, Com. ACM 17-7 (July 74)

/6/ P.J. DENNING
"Fault tolerant operating systems"
Purdue University, India - ACM Computing Surveys, vol.8, n°4 (Dec. 1976)

/7/ T.A. LINDEN
"Operating system structures of support security and reliable software"
ACM Computing Surveys, vol. 8, n°4 (dec.76)

/8/ W. WULF & al
"Hydra : the kernel of a multiprocessor system"
Communications ACM 17-6 (june 76)

/9/ R.M. NEEDHAM, R.D.M. WALKER, A.D.BIRRELL
"The CAP system"
Three papers at the 6th ACM Symp. on O.S. Principles (nov. 77)

/10/ J.B. GOODENOUGH
"Exception handling : issues & a proposed notation"
Communications of ACM, (Dec. 75)

/11/ R. LEVIN
"Program structures for exceptional condition handling"
Ph. D. Dissertation, Carnegie-Mellon University, Pittsburgh (june 77)

/12/ F. CRISTIAN
"Le traitement des exceptions dans les programmes modulaires"
Thèse de docteur ingénieur, IMAG Grenoble (Sept.79)

/13/ B.H. LISKOV, A. SNYDER
"Exception handling in CLU"
IEEE Transact. of Software Engineering, Vol. SE-5, n° 6, (nov. 79)

/14/ J.P. BANATRE
"Control transfer disciplines for exception handling"
IFIP Working Conference on "Reliable Computing and Fault Tolerance in the 1980's", London, (sept. 1979)

/15/ C.A.R. HOARE
"Monitors : an operating system structuring concept"
The Queen's Univ. of Belfast, Com. ACM 17-10 (Oct.74)

/16/ N. WIRTH
"Programming in Modula-2"
Springer-Verlag, Berlin Heidelberg, New-York 1982

/17/ J. VERHOFSTAD
"Recovery techniques for data base systems"
ACM Computing Surveys, vol. 10, n°2 (june 78)

/18/ B. RANDELL
"System structure for software fault tolerance"
University of Newcastle Upon Tyne, IEEE Trans. on Software Engineering, (June 75)

/19/ D.L. RUSSEL
"Process backup in producer-consumer systems"
Proc. Symp. on O.S. principles, operating syst. rev. 11,5 (Nov. 77)

OPTIMAL DETECTION OF SENSOR FAILURES IN FLIGHT CONTROL SYSTEMS USING DETERMINISTIC OBSERVERS

by
Norbert Stuckenberg
Institut für Flugführung, DFVLR, Flughafen
D-3300 Braunschweig-Germany

SUMMARY

A failure detection scheme for sensors of a flight control system is presented. Based on analytic redundancy a duplex sensor configuration provides the fail-operational capability of a conventional triplex sensor system. This is achieved by using deterministic observers. It is shown how the performance of the failure detection scheme can be determined. With respect to this performance criterion the optimal observer is derived. Thus, the performance eventually achievable by an optimal failure detection scheme is also described. The operational feasibility of the proposed concept is demonstrated by flight test results.

1. INTRODUCTION

Reliability plays a vital role in flight control systems of today and in those of the future. Particularly the most attractive control concepts such as artificial stability for the enhancement of maneuverability and flight economy require control systems of extremely high reliability. Traditionally this requirement is fulfilled by using multiple devices in the vital parts of the control systems. For example, it is necessary to triplicate the hardware and to add a majority voting mechanism in order to achieve a fail-operational capability.

However, it is obvious that the conventional hardware redundancy has many disadvantages due to costs, weight, volume, energy consumption, failure rates and maintenance costs. Therefore it is reasonable to look for alternative methods which reduce the necessary efforts in hardware without loss of reliability.

For the sensor part of a flight control system a starting point for the reduction of hardware is the fact that the signals which have to be measured are output signals of one single plant. The plant itself is given by the aircraft motion described by the flight mechanics equations. As analytically given by the equations, the plant outputs are internally coupled. These relations can be used for reliability purposes. Herewith, the hardware redundancy can be replaced by the so-called analytic redundancy.

The basic tools for utilizing the analytic redundancy are filters and observers. Their algorithms have to be implemented into the signal processing part of the control system. Thus sensor hardware is replaced by computer software.

In recent years several methods have been developed following the described common idea on different ways |1|, |2|, |3|, |4|, |5|, |6|. In this paper a concept |6| is proposed omitting the third sensor of a triplex sensor system. Nevertheless, the fail-operational capability is to be maintained. This is achieved by analytic redundancy performed in deterministic observers.

2. SYSTEM OVERVIEW

2.1 The closed loop control system

Fig. 1 shows the general structure of the complete system partitioned into the closed loop control system on the one hand and into the failure detection on the other hand.

The closed loop control system consists of the plant, a duplex sensor configuration, a sensor switching device and the controller. The plant is given by the flight dynamics of the aircraft, forced by the control signals and the disturbances such as air turbulence. The output of the plant is defined such that the resulting vector y contains as much information about the plant state as it is needed for the control problem. The output signals are measured by sensors which are put into a duplex sensor configuration resulting in two identical sensor packages (SP). Hence, either SP includes one sensor of every sensor type (ST). As far as no sensor failure occurs the resulting two measurement vectors z_1 and z_2 are identical.

These two sets of measurement signals are fed into a controller via a sensor switching device. In this device the output signals of a failed sensor is cut off. Output signals of good sensors only can pass. This is shown in fig. 1 for a sensor type i according to the following scheme:

No sensor failure:	The sum of both sensor outputs z_{i1} and z_{i2} multiplied with the factor 0.5 is fed to the controller.
Sensor i in SP1 failing:	Only the signal z_{i2} of the corresponding good sensor of SP2 is fed to the controller.
Sensor i in SP2 failing:	Only the signal z_{i1} of the corresponding good sensor of SP1 is fed to the controller.

This scheme applies to each of the existing sensor types (in fig. 1 shown for ST i only). The switch command signal is generated within the failure detection logic as part of the failure detection.

The controller is supposed to be designed such that the closed loop control system meets the usual requirements, i.e. good response to command inputs and an effective suppression of disturbances over a certain area of the flight envelope.

2.2 The failure detection

The failure detection part of the complete system shown in fig. 1 consists of two identical deterministic observers and a failure detection logic. The observers operate in the usual way. A mathematical model of the plant is driven by the control signals already mentioned as input signals to the real plant. The output $\hat{y}$ of the model is an estimate of the real plant output y. The difference vector between the measured output z_1 (or z_2 resp.) and the estimated output $\hat{y}_1$ (or $\hat{y}_2$ resp.) in this paper called the observer difference is fed back to the plant model via the observer gain matrix.

The two observers shown in fig. 1 are used to provide the information about whether a sensor has failed in SP1 or SP2. This capability is evident from the following state equations.

Let the plant be given as the linear system in the usual notation:

$$x = Ax + Bu + w \tag{2.1}$$

$$y = Cx \tag{2.2}$$

Failures defined as additive signal vectors v_1 and v_2 respectively are given as:

$$z_1 = y + v_1 = Cx + v_1 \tag{2.3}$$

$$z_2 = y + v_2 = Cx + v_2 \tag{2.4}$$

Given an observer gain matrix K the state equation of observer 1 is:

$$\dot{\hat{x}}_1 = A\hat{x}_1 + KCx + Kv_1 + Bu - KC\hat{x}_1 \tag{2.5}$$

and of observer 2:

$$\dot{\hat{x}}_2 = A\hat{x}_2 + KCx + Kv_2 + Bu - KC\hat{x}_2 \tag{2.6}$$

Using the estimation errors defined as:

$$m_1 = x - \hat{x}_1 \tag{2.7}$$

$$m_2 = x - \hat{x}_2 \tag{2.8}$$

the observer dynamic is reduced to the state equation:

$$m_1 = (A - KC)\, m_1 - Kv_1 + w \tag{2.9}$$

$$n_1 = z_1 - \hat{y}_1 = Cm_1 + v_1 \tag{2.10}$$

and with respect to observer 2:

$$\dot{m}_2 = (A - KC)\, m_2 - Kv_2 + w \tag{2.11}$$

$$n_2 = z_2 - \hat{y}_2 = Cm_2 + v_2 \tag{2.12}$$

Equations (2.9) to (2.12) show that the failure vector v_1 affects the difference signal n_1 of observer 1 only whereas the failure vector v_2 correspondends to the observer difference n_2 only. Secondly, the influence of the disturbance vector w is identical in either observer difference. Since this disturbance influence is supposed to be limited, the maximum disturbance response can be used as thresholds for the failure detection task in the following way.

Given are the thresholds n_{iT} of each element n_{i1} and n_{i2} of the vector n_1 and n_2. If one or more elements n_{i1} of observer 1 increase beyond their respective thresholds n_{iT}, it is certain that a sensor failure has occurred in sensor package 1. Equivalently an occurrence of a sensor failure in SP2 can be seen from the response of the observer difference vector n_2.

The final localization of a failed sensor within the SP1 (or SP2) is accomplished by the simple comparison between the output signals of the corresponding sensors of the same type. This part of the failure detection is identical to that used in the conventional hardware redundancy concepts.

Based on these relations the failure detection logic is defined by the following statements:

$$|z_{i1} - z_{i2}| \neq 0 \quad \text{: sensor failure in ST i} \tag{2.13}$$

$$(|n_{11}| > n_{1T}) \vee (|n_{21}| > n_{2T}) \vee \ldots\text{: sensor failure in SP1} \tag{2.14}$$

$$(|n_{12}| > n_{1T}) \vee (|n_{22}| > n_{2T}) \vee \ldots\text{: sensor failure in SP2} \tag{2.15}$$

Using these statements a failed sensor is clearly localized. Then, switch-off actions are taken as described in section 2.1.

In conventional applications observers are used to provide additional information about the plant state in order to improve the control performance in a certain optimal way. Differing from those systems which integrate the observer into the closed loop the present concept uses the observer in an open loop configuration. This offers the opportunity to attach at either sensor package further observers in addition to the single one represented in fig. 1. These additional parallel observers would support each other with respect to their common failure detection task.

Up to now the derivations have been made assuming that there are no deviations between the plant and the plant model inserted in the observers. However, in the real world deviations always exist. In the proposed concept these model errors act in the same way as the disturbances considered so far, again due to the fact that the observers operate in an open loop configuration. Hence, limited model errors have a limited effect on the observer differences, too. For instance, they do not yield instabilities as they could do if an observer is used within a closed loop configuration. Consequently, in the remainder of this paper the disturbance vector w is supposed to include both the real external disturbances and the inner disturbances due to model errors.

3. THE FAILURE DETECTION PERFORMANCE OF A GIVEN OBSERVER

3.1 Performance definition

The task of the concept described is to guard the control system against the effect of sensor failures. The statements of eq. (2.13) show that in principal it is no problem to identify the type of the faulty sensor. However, it is the intrinsic problem of the proposed concept to indicate whether SP1 or SP2 contains the faulty sensor. This is due to the nonzero thresholds of the observer differences which result in the possibility of a sensor failure also reaching large values until being identified in one of the sensor packages. This means that the control system can undergo deviations from its desired path, too. Hence, it is obvious, also from fig. 1 and from the state equations, that a sensor failure causes two competing effects:

- The closed loop control system can be driven away from the desired nominal trajectory.
- At least one of the observer differences of the respective observer is driven beyond its threshold, by that indicating the faulty SP.

For a given system, i.e. a control system with given feedback gains and observers with fixed observer gain matrices, the identifiability of a sensor failure depends on the one hand on the magnitude of the disturbance thresholds and on the other hand on the magnitude and the signal character of the sensor failure itself. For both signals worst-case assumptions have to be adopted. As far as the disturbances are concerned, the worst-case requirement is already fulfilled by defining the disturbance threshold as the maximum disturbance response. As far as the sensor failure itself is concerned, no assumptions and preconditions of any kind are permitted. The identification system has to cover all kinds of sensor failures, particularly the most unfavorable one. This worst-case failure signal is the one which causes a great deviation in the control system output and has a low sensitivity with respect to the observer differences. Thus, the performance of the detection system with respect to a failure in a certain sensor has to be defined using the following value: The maximum response of a control system output due to that failure input which is not able to drive one of the observer differences beyond its threshold. The lower this particular deviation of the control system the better the detection performance.

3.2 Transformation to an optimal control problem

Now, the performance of the failure detection has to be calculated. According to the previous section it is necessary to weigh the sensor failure response in the control system and the sensor failure response in the observers against one another.

These two competing relations can be derived from the following concise representation of the relevant state equations.

$$\begin{bmatrix} \dot{m} \\ \dot{x} \end{bmatrix} = \left[\begin{array}{c|c} A - KC & o \\ \hline o & A - BRC \end{array}\right] \begin{bmatrix} m \\ x \end{bmatrix} - \left[\begin{array}{c} K_i \\ \hline 1/2(BR)_i \end{array}\right] v_i \qquad (3.1)$$

$$n_i = C_i m + v_i \qquad (3.2)$$

$$n_j = C_j m \qquad , j \neq i \qquad (3.3)$$

$$y_k = C_k x \qquad , k = i,j \qquad (3.4)$$

In this representation the control law of fig. 1 is chosen to be a linear feedback given by the gain matrix R. Furthermore, these equations apply to failures v_i from either SP. Consequently, the indices 1 and 2 for identifying the two SPs and the two observers are omitted. Henceforth, an index identifies the element of a vector or the row (column) of a matrix respectively. A block diagram representation of equ. (3.1) to (3.4) is given in fig. 2a. Using this figure the worst-case sensor failure as input signal to two separate systems will be discussed. Once the disturbance thresholds n_{iT} and n_{jT} are fixed, the worst-case failure input v_i is that signal which forces the control system output y_k to the maximum response y_{kmax} under the constraint that the simultaneously forced observer differences n_{iT} and n_{jT} remain within their thresholds.

Though forced by the common input signal v_i the direct relation between the output signals of two separate systems is essential for the determination of the detection performance. This direct relationship can be derived by proceeding from the common description of the two systems in equ. (3.1) to (3.4). The output equ. (3.2) is transformed into

$$v_i = n_i - C_i m \tag{3.5}$$

and v_i used in equ. (3.1) again. What results is a transformed system

$$\begin{bmatrix} \dot{m} \\ - \\ \dot{x} \end{bmatrix} = \left[\begin{array}{c|c} A - KC + K_i C_i & o \\ \hline 1/_2(BR)_i C_i & A - BRC \end{array}\right] \begin{bmatrix} m \\ - \\ x \end{bmatrix} - \begin{bmatrix} K_i \\ --- \\ 1/_2(BR)_i \end{bmatrix} n_i \tag{3.6}$$

$$n_j = C_j m \quad , \qquad j \neq i \tag{3.7}$$

$$y_k = C_k x \quad , \qquad k = i,j \tag{3.8}$$

The output n_i of the original system is now defined as the input of the transformed system. This is possible since in the output equ. (3.2) the input v_i was contained in a direct superposition. The equivalence of both representations also becomes apparent from the block diagrams shown in fig. 2a and fig. 2b.

The problem of the greatest effect of a sensor failure which is still just undetected can now be described more simply with the aid of the block diagram of the transformed representation: What is sought is the maximum value of an output y_k which can be generated by the now limited input n_i. Within the limitation n_{iT} the input signal n_i can assume any given signal behaviour. Additionally, the other observer differences n_j have to be kept within their respective limitations n_{jT}. The time for reaching the maximum value y_{kmax} is unlimited.

Thus, the problem of the greatest effect of an undetectable sensor failure and with it the failure detection performance is reduced to an optimal control problem. The solution methods known for problems of this nature can be applied. Since for such problems the solutions cannot be stated in the form of complete formulae usually they must be found specifically for each case. The solution process for a concrete example is carried out in |6|. Only the principal characteristics of a solution will be stated at this point.

3.3 The worst-case failure signal

For a qualitative example fig. 3 shows the optimal time histories which lead to a maximum value y_{kmax}, given an input n_i which is constrained by the threshold n_{iT}. For a system with a limited input the optimal control theory (maximum principle) requires positive and negative maximum values as input signals. Hence, the rectangular function n°_i in fig. 3 is found to be optimal in terms of optimal control. I.e., from the failure detection point of view it is the most adverse one. The switching instants are determined by the switching variable q alternating its sign. The system output y_k is forced by the optimal control signal n°_i to reach the maximum value when the final time t_f goes to infinity. However, in pratical applications t_f remains finite. As it can be deduced from optimal control theory the optimal input signal n°_i is not periodic for systems of an order $\geq$ 2. Again, in practice the rectangular function n°_i oscillates with nearly constant frequency.

Once the worst-case input n°_i and its response m°_i of the transformed system of fig. 2 is determined, the equivalent worst-case failure signal v°_i itself is given by the relation

$$v^\circ_i = n^\circ_i - C_i m^\circ \tag{3.9}$$

It turns out to be a discontinuous function as shown in fig. 3. In terms of the original system representation of fig. 2 this input signal v°_i would create the greatest response in the output y_k of the control system without the respective observer difference n_i exceeding its threshold n_{iT}.

In fig. 3 the threshold n_{jT} of the other observer difference n_j was not effective since the signal n_j remained within this limitation. However, given a threshold n_{jT} lower than the maximum values of n_j shown in fig. 3 the most adverse solution has to shift. As derived in |6| the worst-case failure signal v_i is determined now such that the observer difference n_i becomes a rectangular function with a shifted frequency. This new switching frequency is defined such that the other observer difference n_j just reaches its threshold n_{jT}. The maximum values $\pm$ n_{iT} of the rectangular observer difference n_i are retained. Of course, the maximum output y_{kmax} of the control system is smaller now, i.e. the performance of the failure detection has improved.

A chance to lift the performance furthermore can be used by attaching an additional filter at the observer difference n_j as illustrated in fig. 4. The filter output n_{jF} is implied into the failure detection procedure in the same way as the original observer differences n_i and n_j are. The filter itself has to have a sensitivity with respect to the just discussed frequency. But it is as insensitive as possible at other frequencies. Hence, it can be assumed that the response to disturbances is small. Thus, the new threshold n_{iFT} defined as the maximum response to these inputs is small, too. The worst-case failure signal v_i has to be changed again in such a way that the filter output n_{iF} also just reaches its threshold n_{iFT}. Consequently, the maximum output y_{kmax} of the control system has decreased, i.e. the failure detection has been improved.

A necessary condition for detecting a sensor failure v_i via the observer difference n_j or the filter output n_{iF} is that a sufficiently high coupling between the failure signal v_i and the other observer difference n_j exists particularly in the worst-case situation. This problem is treated in detail in |6|. At this point it shall be discussed only with regard to the transformed system representation of fig. 2b.

Let the system matrix $(A - KC + K_iC_i)$ be unstable such that the response C_im to even small inputs n_i becomes large. Then, also the control system output y_k is forced to large values via the element $(BR)_i$. This critical situation however cannot occur if via the system matrix $(A - KC + K_jC_j)$ the output n_j is driven to high values, too. Because this signal n_j has to remain within its threshold n_{jT} the signals C_jm and y_k are also kept on a lower level. However, in this case it is necessary that a sufficiently high coupling exists between C_im and n_j. In other words, the output C_im has to be observable from the output n_j. Hence, it is a matter of the uncontrolled plant, i.e. the system matrix A and the output matrix C. Their properties eventually define the range of applicability of the proposed concept.

4. THE OPTIMAL OBSERVER SUBJECT TO THE SENSOR FAILURE DETECTION PROBLEM

4.1 Sensor failures, disturbances and performance criteria

In the previous chapter we have analyzed the performance of the failure detection on the condition that the observer dynamic is given. However, the proposed concept fixes the observer structure only as shown in fig. 1. The elements of the observer gain matrix are still free to be chosen. Hence, two principal questions arise:

- How has the observer gain matrix to be chosen in order to yield the optimal observer with respect to the failure detection task?

- Which is the eventual performance of this optimal observer?

Since the structure of the observer is that of a Kalman-filter, one could think of using its optimization techniques for the design of the observer in the present case, too. However, a comparison list concerning the design criteria and further assumptions of the two cases shows that there are considerable differences.

Kalman-filter	Sensor failure detection observer
Sensor failures (measurement noise resp.)	
stochastic signal with zero mean, white noise spectral density and Gaussian distribution	deterministic, discontinuous, aperiodic signal according to section 3.3 and fig. 3
System noise	
stochastic signal with zero mean, white noise spectral density and Gaussian distribution	those deterministic and/or stochastic disturbances which cause the maximum response of the observer differences
Performance criterion	
minimum mean-square-error between state and estimated state	minimum amplitude of a control system output due to a not detected sensor failure

Table 1: Comparison of the Kalman-Filter and the sensor failure detection observer

This comparison given in table 1 shows that the design of the observer as a Kalman-filter cannot be optimal with respect to the sensor failure detection task. As far as the measurement noise is concerned, the two signals differ not only in the way they appear but also in the way they are determined. In the Kalman-filter case the measurement is an independent signal whereas in the present case the sensor failure is dependent on the observer dynamic itself as was found in chapter 3. Thus, under these difficult conditions it seems to be almost impossible to find the observer gain matrix without easing off the strict assumptions given in table 1.

Hence, a suboptimal solution is sought by replacing the assumptions stated in table 1 by the following approximations.

Sensor failure signals:

Fig. 3 has shown that in practice the worst-case failure signal v°_j can be assumed as a periodic signal. Furthermore, its discontinuities ought to be neglected. Thus, the approximate worst-case failure signal shall be determined as a stationary harmonic signal. Its frequency cannot be determined yet because at this point of the considerations it is still a matter of the observer dynamic.

Sytem noise:

The deterministic part of the disturbances is approximated by a harmonic signal. As far as the stochastic part is concerned it is assumed that it can also be approximated by an equivalent harmonic signal. The justification will be given in the next section when the frequencies of both the sensor failures and the disturbances are determined.

Performance criterion:

The ampltiude of the failure signal just not detectable by the observer shall become as small as possible. This is a slight modification of the performance criterion used in the previous chapter.

Now, the overall design procedure of the observer can be considered as a two step approach. First, the optimization can be carried out according to the approximations given here. Secondly, the final performance of the failure detection has to be assessed on the basis of the more rigorous conditions established in chapter 3.

4.2 The failure signal frequency

In the foregoing section it has been shown that the optimization of the observer is a deterministic problem. Both the failure signals and the disturbances are supposed to be harmonic signals. The frequency of the sensor failure signal has to be determined under the constraint that the relationship between the failure signals and the observer dynamic covers the most adverse conditions. Let us now discuss this problem by using an example based on a second-order system.

For a 2nd-order system the failure detection problem is reduced to the block diagram represented in fig. 5. In this example the following assumptions are established:

- The output matrix is the unity matrix: $C = I$
- Failure in sensor 1: failure signal v_1
- Failure detection at observer difference n_2

Thus, the respective state equation becomes:

$$\begin{bmatrix} \dot{m}_1 \\ \dot{m}_2 \end{bmatrix} = \begin{bmatrix} A - K \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \end{bmatrix} - \begin{bmatrix} K_1 \end{bmatrix} v_1 + \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \tag{4.1}$$

The disturbances w_1 and w_2 are independent signals. Hence, it is sufficient to carry out the derivations with respect to the signal w_1 only and to apply the results to the other signal w_2 thereafter.

From equ. (4.1) Laplace-transformed transfer functions can be derived:

Sensor failure response:

$$n_2(s) = \frac{\begin{bmatrix} sI - (A - K) \end{bmatrix}_{adj} \begin{bmatrix} - K_1 \end{bmatrix} v_1(s)}{\det \left\{ sI - (A - K) \right\}} = \frac{Z_v(s)}{N(s)} \; v_1(s) \tag{4.2}$$

Disturbance response:

$$n_2(s) = \frac{\begin{bmatrix} sI - (A - K) \end{bmatrix}_{adj} \begin{bmatrix} 1 \\ 0 \end{bmatrix} W_1(s)}{\det \left\{ sI - (A - K) \right\}} = \frac{Z_w(s)}{N(s)} \; w_1(s) \tag{4.3}$$

Now, the input signals are assumed to be harmonic oscillations. The respective frequencies ω_v and ω_w are first assumed to be arbitrary but distinct, i.e.:

$$\omega_v \neq \omega_w \tag{4.4}$$

The amplitudes of the stationary harmonic response of the observer difference n_2 are:

$$|n_2(j\omega_V)| = \frac{|Z_V(j\omega_V)|}{|N(j\omega_V)|} |v_1(j\omega_V)| \tag{4.5}$$

and

$$|n_2(j\omega_W)| = \frac{|Z_W(j\omega_W)|}{|N(j\omega_W)|} |w_1(j\omega_W)| = n_{2T} \tag{4.6}$$

In equ. (4.6) it is also stated that the stationary amplitudes $|n_2(j\omega_W)|$ representing the maximum disturbance response have to be defined as the threshold n_{2T}.

The detection of the harmonic failure v_1 is established as soon as the harmonic failure response $|n_2(j\omega_V)|$ reaches the threshold n_{2T}, i.e.:

$$|n_2(j\omega_V)| = n_{2T} \tag{4.7}$$

Using equ. (4.5) and equ. (4.6) in equ. (4.7) yields a direct relationship between the disturbance w_1 and the just detected sensor failure v_1 (or the maximum just not detectable sensor failure v_1).

$$|v_1(j\omega_V)| = \frac{|Z_W(j\omega_W)| \quad |N(j\omega_V)|}{|Z_V(j\omega_V)| \quad |N(j\omega_W)|} |w_1(j\omega_W)| \tag{4.8}$$

Applying the performance criterion of the previous section the amplitudes $|v_1(j\omega_V)|$ have to be kept as small as possible. Equ. (4.8) offers the chance to achieve this objective via the common denominators $N(j\omega_V)$ and $N(j\omega_W)$ of the transfer functions given in equ. (4.2) and equ. (4.3). If the eigendynamic of the observer is chosen like that shown in fig. 6 then the fraction $|N(j\omega_V)|/|N(j\omega_W)|$ in equ. (4.8) becomes arbitrarily small. In physical terms: The observers sensitivity with respect to sensor failures is chosen as much higher than that with respect to the disturbances.

Fig. 6 also demonstrates the favorable effect of a bandpassfilter already suggested in section 3.8 and fig. 4. This filter suppresses the effect of the disturbances and further amplifies the effect of the sensor failures.

However, the filter effects just described are successful only if, actually, the frequencies of the sensor failures and the disturbances are distinct as presumed in equ. (4.4). Now, it has to be recalled that the observer system has to be able to operate under the most adverse conditions. From fig. 6 it is obvious that this case is given if the two frequencies coincide, i.e. if

$$\omega_V = \omega_W \tag{4.9}$$

Under this constraint the detection of sensor failures is no longer performed via the observer eigendynamic because the denominators of the two transfer functions of equ. (4.2) and equ. (4.3) disappear in equ. (4.8). The failure detection performance can be optimized only by manipulating the remaining numerators.

At this point it also becomes clear that it is justified to replace stochastic disturbances by an equivalent harmonic signal since the latter represents the more difficult problem from the failure detection point of view. The broad power spectrum of a stochastic signal could be covered by one or more parallel bandpassfilters in the way demonstrated in fig. 6 such that a high failure detection performance results. On the other hand a narrow power spectrum can be approximated by a harmonic signal as proposed in section 4.1.

From the discussion presented it follows that first, an observer has to be optimized under the most difficult conditions of coinciding frequencies. The resulting failure detection performance is the crucial one. Secondly, other failure frequencies possible are covered by bandpassfilters or by another parallel observer provided with an eigendynamic like that shown in fig. 6. This case will not be considered furthermore since the respective detection performance is always higher than that of the crucial case.

The foregoing derivations have been made with respect to the first disturbance signal w_1 of a 2nd-order system as used in equ. (4.1). Obviously, there is no problem to apply the principal results to the disturbance signal w_2, too, or to further input signals when a system of higher order is given.

4.3 The optimal observer

The optimal observer provides the best performance, i.e. the smallest not detectable sensor failure under the worst conditions described in the previous section. Also the value of this ultimate performance achievable is of interest. Again, the problem will be solved based on a 2nd-order system first. Then, the results are generalized with respect to higher order systems.

Let the example be given as represented in fig. 5, i.e.:

- Second-order system
- Failure in sensor 1: failure signal v_1
- Failure detection at observer difference n_2

Given the constraint of equ. (4.9) the relationship between the sensor failure v_1 and the disturbances w_1 and w_2 in Laplace-transformed representation becomes:

$$v_1 = \frac{\det \left\{ \left[(sI - (A - K)) \right]_1 \,\vdots\, \begin{matrix} w_1 \\ w_2 \end{matrix} \right\}}{\det \left\{ \left[(sI - (A - K)) \right]_1 \,\vdots\, - K_1 \right\}} \tag{4.10}$$

Using an elementary determinant transformation equ. (4.10) becomes in detail:

$$v_1 = \frac{\det \begin{Bmatrix} s - (a_{11} - k_{11}) & \vdots & w_1 \\ - (a_{21} - k_{21}) & \vdots & w_2 \end{Bmatrix}}{\det \begin{Bmatrix} s - (a_{11} - k_{11}) & \vdots & s - a_{11} \\ - (a_{21} - k_{21}) & \vdots & - a_{21} \end{Bmatrix}} \tag{4.11}$$

It should be noted that these dynamic relations between the sensor failure and the disturbances are not dependent on the second column K_2 of the observer gain matrix K. In more general terms: The gain matrix column corresponding to that observer difference performing the failure detection does not affect the failure-disturbance relations.

Developing the relation of equ. (4.11) into the transfer functions $(v_1/w_1)(s)$ and $(v_1/w_2)(s)$ one obtains:

$$v_1(s) = \left(\frac{v_1}{w_1}\right)(s) \quad w_1(s) + \left(\frac{v_1}{w_2}\right)(s) \quad w_2(s) \tag{4.12}$$

The disturbances w_1 and w_2 have to be considered as independent signals. Therefore, when minimizing v_1 the two portions in equ. (4.12) have always to be added. Fortunately, this condition is fulfilled by applying harmonic signals again. By that the transfer functions turn over to Bode amplitude plots which by definition represent positive values only.

Now, the harmonic, stationary amplitudes of the sensor failure v_1 as response to the harmonic inputs w_1 and w_2 have to be minimized by modifying the column K_1 of the observer gain matrix K. Using the abbreviations

$$f_{ij} = a_{ij} - k_{ij} \tag{4.13}$$

two limiting cases can be identified.

Case $f_{11} >> f_{21}$:

The amplitudes v_1 due to the disturbance w_1 tend to zero, i.e.:

$$\lim_{f_{11} >> f_{21}} \left| \frac{v_1}{w_1} \right| (\omega) = 0 \tag{4.14}$$

However, at the same time the response due to the disturbance w_2 tend to a limiting amplitude plot as shown in fig. 7a. This means that in practice for finite frequencies a limit is given by:

$$\lim_{f_{11} >> f_{21}} \left| \frac{v_1}{w_2} \right| (\omega) = \frac{1}{a_{21}} \tag{4.15}$$

Case $f_{21} >> f_{11}$:

This case is the reverse to the previous one. As shown in fig. 7.b the first sensor failure-disturbance relation (v_1/w_1) tends to the limiting amplitude plot of the transfer function $1/(s-a_{11})$, i.e.:

$$\lim_{f_{21} >> f_{11}} \left| \frac{v_1}{w_1} \right| (\omega) = \left| \frac{1}{j\omega - a_{11}} \right| \tag{4.16}$$

whereas simultaneously the second relation tends to zero, i.e.

$$\lim_{f_{21} \gg f_{11}} \left| \frac{v_1}{w_2} \right| (\omega) = 0 \tag{4.17}$$

These results are summarized and schematized in table 2

	$f_{11} \gg f_{21}$	$f_{21} \gg f_{11}$
$\lim\left(\frac{v_1}{w_1}\right)$	0	$\frac{1}{s - a_{11}}$
$\lim\left(\frac{v_1}{w_2}\right)$	$\frac{1}{a_{21}}$	0

Table 2: Limits of the sensor failure-disturbance relations of a 2 nd-order system

Table 2 demonstrates that, obviously, it is not possible to bring both the relations down to zero simultaneously.

Two results should be emphasized:

- The sensor failure-disturbance relations are independent of the column K_2, that is the column corresponding to the observer difference which is supposed to yield the failure detection.

- The result of the optimization process, i.e. the optimized sensor failure-disturbance relations, is entirely independent of the observer gain matrix but determined by the column A_1 of the plant matrix A only. This column A_1 corresponds to that sensor which is affected by the failure.

4.4 Generalizations

The results derived in the previous sections with respect to a 2nd-order system are now generalized without proof and applied to a system of higher order.

Consider the following assumptions:

- Plant system matrix: A

- System order: n

- Failure in sensor i: v_i

- Failure detection at observer difference n_j

Then, the following limits of the sensor failure-disturbance relations are possible:

$$v_i = \frac{w_1}{a_{1i}} \vee \frac{w_2}{a_{2i}} \vee \ldots . \vee \frac{w_i}{s-a_{ii}} \vee \ldots . \vee \frac{w_n}{a_{ni}} \tag{4.18}$$

Given the amplitudes of the above disturbances the smallest value of the quotients offered in equ. (4.18) can be elected. This value represents the eventual failure detection performance achievable, i.e. the smallest amplitude of the failure signal v_i just not detected. The frequency of this failure signal is identical to the frequency of that disturbance signal elected from equ. (4.18).

In order to achieve this final result the observer gain matrix has to be chosen similarly to the cases discussed with respect to a 2nd-order system. Again, the column K_j corresponding to the failure detecting observer difference n_j is not available to this optimization procedure.

However, this observer optimized with respect to a good failure detection performance is not necessarily supposed to be optimal from the conventional design point of view since its eigendynamic has not been a matter of consideration so far. But from a practical point of view the eigendynamic cannot be neglected. For instance, a poor stability though not affecting the stationary sensor failure-disturbance relations is of no use. Hence, at this point the column K_j of the observer gain matrix K can be used finally in order to establish a good eigendynamic.

5. APPLICATION TO A COMPLETE FLIGHT CONTROL SYSTEM FOR A TRANSPORT AIRCRAFT

The failure detection concept was applied to a given flight control system for a transport aircraft. The objective of this experiment was, first of all, to demonstrate the operational feasibility in a realistic environment. Hence, it was not necessary to lay emphasis on the observer design as the optimal one described in the previous chapter.

5.1 The flight control system

Fig. 8 shows a simplified block diagram of the flight control system assigned to the failure detection experiment. The objectives of this control system are on the one hand a good response to the command signals defined as the command vector.

$$y_c = \begin{bmatrix} \text{altitude command} & H_c \\ \text{speed command} & V_c \\ \text{bank angle command} & \phi_c \end{bmatrix} \qquad (5.1)$$

and on the other hand a sufficient suppression of the disturbances w.

A good command response is achieved essentially by a careful design of the feedforward signal path, given a well damped eigen behavior of the closed loop system by choosing reasonable feedback gains. Then, this latter property provides also for a sufficiently low disturbance response.

In order to achieve these design objectives it is necessary, first, to control the plant by the control vector u defined as

$$u = \begin{bmatrix} \text{elevator deflection} & \eta \\ \text{thrust} & F \\ \text{aileron deflection} & \xi \\ \text{rudder deflection} & \zeta \end{bmatrix} \qquad (5.2)$$

and, secondly, to get the information about the plant state via the measurement vector z, defined as

$$z = \begin{bmatrix} \text{yaw rate} & r \\ \text{roll rate} & p \\ \text{pitch rate} & q \\ \text{bank attitude} & \phi \\ \text{pitch attitude} & \Theta \\ \text{vertical speed} & \dot{H} \\ \text{altitude} & H \\ \text{indicated air speed} & V_{IAS} \\ \text{lateral acceleration} & a_y \end{bmatrix} \qquad (5.3)$$

This flight control system is successfully flight tested with the DFVLR experimental aircraft HFB 320. A detailed description is given in |7|.

The feedforward control loops are independent of the measurement. Hence, they do not interfere with the sensor failure problem. On the other hand the signal path from the measurements via the feedback law to the control vector u is of particular interest for the sensor failure problem and its solution, since via this loop sensor failures have an effect on the plant.

5.2 The test arrangement for the generation, the detection and the isolation of sensor failures

The failure detection scheme needs a duplex configuration as represented in fig. 1. But in the experimental flight control system only a simplex sensor system was available. Therefore a slight modification of the original concept became necessary for test purposes. This is shown in fig. 9.

A single sensor package was used for the flight control test program. All measurement signals combined into the measurement vector z are splitted up into two separate signal paths. Into either path additive signals can be fed represented by the vectors v_1 and v_2. Thus failures in sensors of the original two sensor packages are simulated by software. The remainder of the original scheme was not changed.

The intrinsic purpose of the experiment, i.e. the indication in which of the two sensor packages a failed sensor is located, can still be achieved. Only the indication of the type of the failed sensor operates in an unrealistic condition. But as already mentioned this indication implies no innovative aspects since it arises in the conventional hardware redundancy concepts, too.

Fig. 10 shows in detail the arrangement of the sensors and the internal structure of the observers as it was used in the flight tests. For clarification purposes observer 1 and the signal path of the simulated sensor package 1 is represented only. The representation of observer 2 and the SP2 is identical to that shown in fig. 10.

The aircraft motion model used in the observers is nonlinear. This is done in order to keep the estimated outputs as close as possible to the outputs of the real plant. The nonlinearities consist of

- the coupling between the longitudinal and the lateral motion

- some quadratic terms depending on the dynamic pressure

- the thrust depending on Mach number and static pressure.

The differences between the measurement signals and the estimated outputs are fed back to the plant via the observer gain matrix. Going beyond this concept shown in fig. 1 and described in the original observer/filter literature additional integral terms are fed back, too. This is done in order to cancel possibly stationary deviations in the respective observer differences completely. The implementation of these integral elements into the observer structure does not raise any problems either theoretically or practically. The observer gain matrix itself is chosen as similar to the gains used in the control system. Though this approach is not stringently optimal as pointed out in chapter 4 it offers some practical advantages. Detailed reasons are given in |6|.

In fig. 10 additional filters are attached to the observer differences n_r and n_θ . These filters operate in the way generally described in section 3.3 and sketched in fig. 4 and fig. 6, i.e. suppression of disturbances and amplification of sensor failures. The actual effect will be discussed using some flight test results.

6. FLIGHT TESTS

Flight tests were conducted using the DFVLR test aircraft HFB 320. First, according to the outlined concept data about the model deviation and real disturbances effects were collected. In the second part the operation of the failure detection and isolation was demonstrated when sensor failure signals were applied.

6.1 Determination of thresholds

The observer differences deviate from zero due to model deviations and disturbances. The response to model deviations becomes high when the control system forces the plant to move dynamically. Therefore, inputs into all 3 command signal paths of fig. 8 are applied successively:

- altitude command with different descent and climb rates

- indicated air speed commands represented as a deceleration procedure

- bank attitude commands.

Some test results are shown in fig. 11 and fig. 12. They are selfexplanatory to a certain extent. Only a few features will be discussed.

Fig. 11: The descent rate is 800 ft/min, the climb rate is 1500 ft/min. Though during the maneuver the plant moves considerably in altitude H, vertical speed $\dot{H}$ and pitch attitude Θ the corresponding observer differences remain small. Only the signal n_θ shows a certain offset which becomes even clearer in the output $n_{\theta F}$ of the attached low pass filter. This effect may be referred to an unprecise modelling of the actual thrust.

Fig. 12: During this flight interval an area of heavy air turbulence was sought and found crossing a strong cumulus.

Based on these data the thresholds for the second part of the flight test were fixed as shown in the first column of table 3.

6.2 Application of sensor failures

As represented in fig. 9 and fig. 10 sensor failures are generated by feeding additive signals into one of the two sensor signal paths. This is done successively for each of the sensors of one sensor package. Because the arrangement of the sensor packages is symmetrical with respect to both the control system and the observers, sensor failures are applied to the SP1 only. The plots of the failure signals are equal but multiplied with an individual factor given in the second column of table 3. The common structure of the failure plots is defined in fig. 13. Since the signal character of the sensor failures is important with respect to their effect on the control system on the one hand and to the failure detectability on the other hand the plot of fig. 13 is partitioned into three different intervals.

Observer difference thresholds	Sensor failure factors
n_{rT} = 1.0 °/s	k_r = 3 °/s
n_{pT} = 2.0 °/s	k_p = 5 °/s
$n_{\dot{\Theta}T}$ = 1.0 °/s	k_q = 5 °/s
$n_{\Phi T}$ = 2.5 °	k_Φ = 10 °
$n_{\Theta T}$ = 1.0 °	k_Θ = 3 °
$n_{\dot{H}T}$ = 2.0 m/s	$k_{\dot{H}}$ = 5 m/s
n_{HT} = 4.0 m	k_H = 50 m
n_{VT} = 2.0 m/s	k_v = 10 m/s
n_{ayT} = 0.5 m/s²	k_{ay} = 5 m/s²
$n_{\Theta FT}$ = 0.5 °	-
n_{rFT} = 0.25 °/s	-

Table 3: Thresholds and sensor failure factors

Interval "10-60 sec": During this period the failure signal increases on a slight slope, thus simulating a small drift of the physical sensor.

Interval "60-70 sec": A stationary offset is represented.

Interval "70-80 sec": A steep decrease to zero is simulating a high drift rate. Thus a failure signal containing higher frequencies is represented.

The sensor failures were applied during a straight level flight in fairly calm air. Thus the effect of the failures becomes clear because they are only little disturbed by air turbulence or command signals. In fig. 14, fig. 15 and fig. 16 test results are shown respective to failures in the Θ-sensor, the V_{IAS}-sensor and the p-sensor. In these figures the plots of the most relevant variables are given showing in the first columns the system responding to the complete failure signal profile without to be interfered by sensor switching. In the respective second columns the operation of the switching logic becomes obvious.

Fig. 14: The additive failure signal in the Θ-sensor clearly shows up in the observer difference n_Θ and similarly in the filter output $n_{\Theta F}$. The aircraft motion itself is almost unaffected by this kind of failure, even during the interval of steep decrease. The failed sensor is detected when the filter output $n_{\Theta F}$ increases beyond its threshold. At this time the switch command signal appears. At the corresponding time instant of fig. 14b the switching operations take place actually.

Fig. 15: The application of the failure to the V_{IAS}-sensor shows results which are very much different from those of the previous failure case. Here, the plant variable V_{IAS} follows the faulty measurement in the V_{IAS}-sensor whereas during the low rate drift interval the corresponding observer difference n_v remains close to zero. This means that the observer difference n_v is unusable for the detection of slow drift rate failures. It indicates only higher drift rate failures as demonstrated during the steep decrease interval. But low drift rates have an effect on the observer difference n_Θ based on the coupling between speed and pitch attitude within the plant and its model. This relation can be used for the failure indication particularly by the filter output $n_{\Theta F}$ since the previously defined disturbance threshold $n_{\Theta FT}$ is small due to the low pass filtering property. Hence, this filter output activates the switching operations in fig. 15b.

Fig. 16: These plots show an example of a failure in a lateral motion sensor. From a systems theory point of view this failure case appears to be similar to that of a failure in the Θ-sensor in fig. 14.

Failures of the remaining sensors can be classified either as similar to the failure of the Θ-sensor (r-, $\dot{\Theta}$-, H-, a_y-sensor) or as similar to the failure of the V_{IAS}-sensor (Φ-, H-sensor). As far as the low drift in the Φ-sensor is concerned, the relation between bank attitude Φ and yaw rate r is utilized amplified in the filter output n_{rF}. However, for the H-sensor an equivalent relation does not exist. Hence, drift failures in this sensor must be declared as undetectable.

7. CONCLUSION

For the sensor part of a flight control system a failure detection concept has been developed. Based on analytic redundancy a duplex sensor configuration achieves the fail-operational capability of a conventional triplex system. Two parallel deterministic observers provide the information about which of the two corresponding sensors is faulty.

Since sensor failures severely influence the safety of flight the failure detection becomes crucial, too. Hence, the performance of the failure detection has been determined with respect to the worst-case failure cases. By using optimal control methods it has been derived that the worst-case failure signals are precisely definable as deterministic, discontinuous and aperiodic signals. However, assuming harmonic failure signals as approximations to those of the exact worst-case the optimization of the observer subject to the failure detection task becomes feasible. Given this optimal observer, the final failure detection performance achievable turns out to be eventually determined by the properties of the uncontrolled plant only.

Flight tests have demonstrated that in principle the failure detection concept is feasible with commonly used sensors.

8. REFERENCES

|1| Clark, R.N., Fosth, D.C., Walton, V.M. Detecting instrument malfunctions in control systems.
IEEE Trans. on Aerospace and Electronic Systems Vol. AES-11 (1975) No. 4, pp. 465-473.

|2| Shapiro, E.Y. Software techniques for redundancy management of flight control systems.
Proc. of AIAA Guidance and Control Conference, Aug. 1976.

|3| Deckert, J.C., Desai, M.N., Deyst, J.J., Willsky, A.S. F8-DFBW sensor failure identification using analytic redundancy.
IEEE Trans. on Automatic Control, Vol. AC-22, 795-803, Oct. 1977.

|4| Montgomery, R.C., Tabak, D. Application of analytical redundancy management to shuttle crafts.
Proc. of the 1978 IEEE Conf. on Decision and Control, San Diego, Cal., Jan. 10-12, 1979.

|5| Labarrere, M., Pelegrin, M., Pircher, M. Automatic recovery after sensor failure on board.
AGARD Conf. Proc. No. 272, Ottawa, Can., May 8-11, 1979.

|6| Stuckenberg, N. An observer approach to the identification and isolation of sensor failures in flight control systems.
ESA-TT-738 (English Translation of DFVLR-FB 81-26).

|7| Adam, V. Leyendecker, H. Control law design for transport aircraft flight tasks.
AGARDograph No. 251.

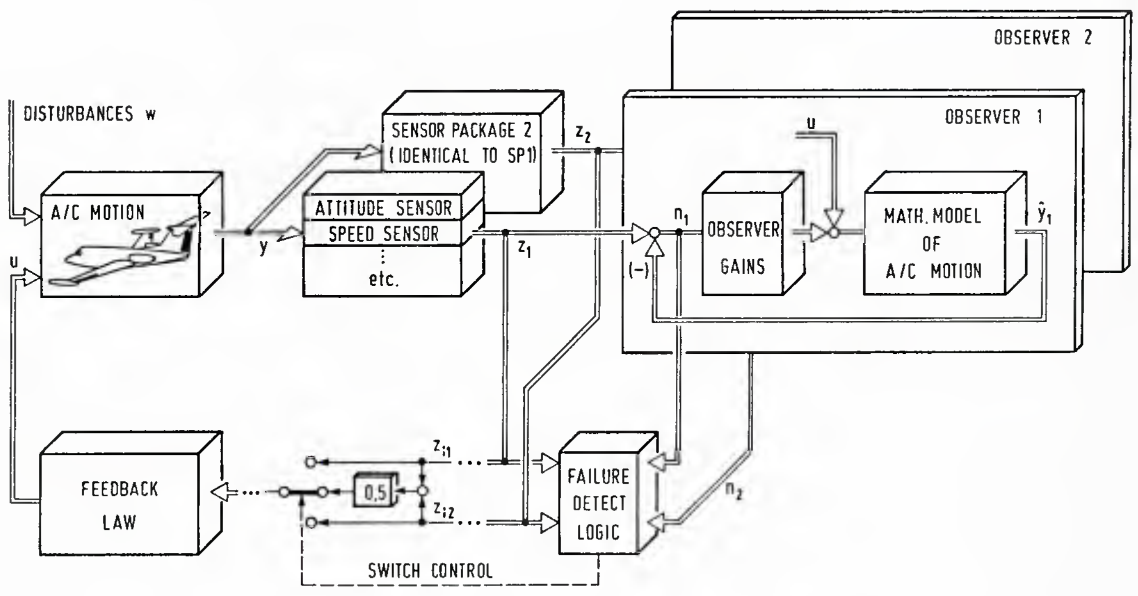
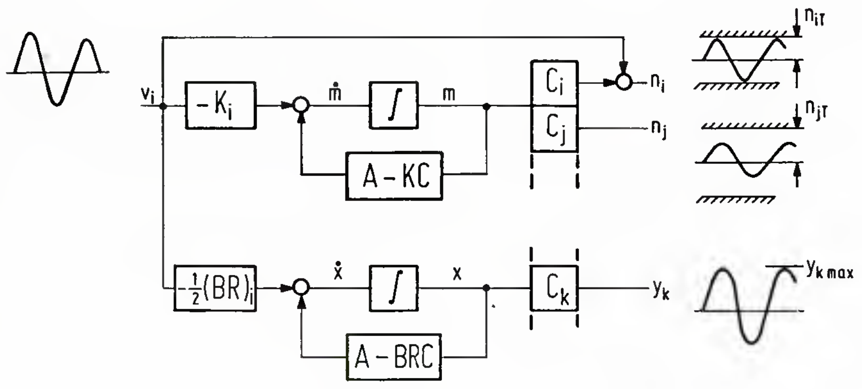
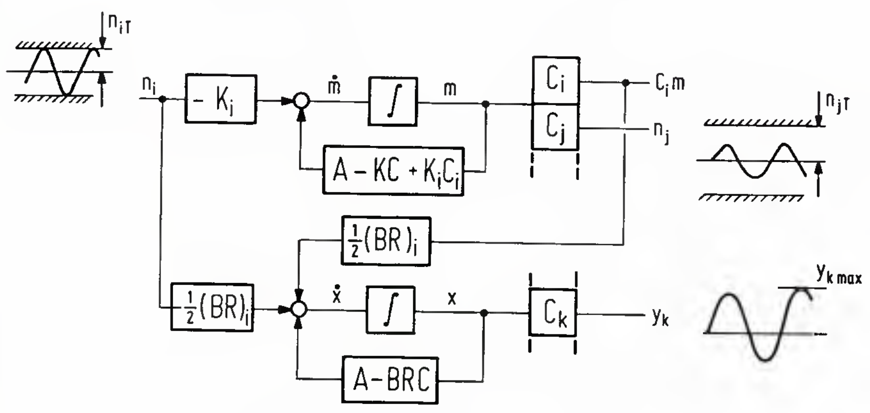


Fig. 1: Flight control system with a fail-op duplex sensor configuration



a: Original block diagram



b: Transformed block diagram

Fig. 2: Relationship between sensorfailure v_i, observer difference n_i and control system output y_k

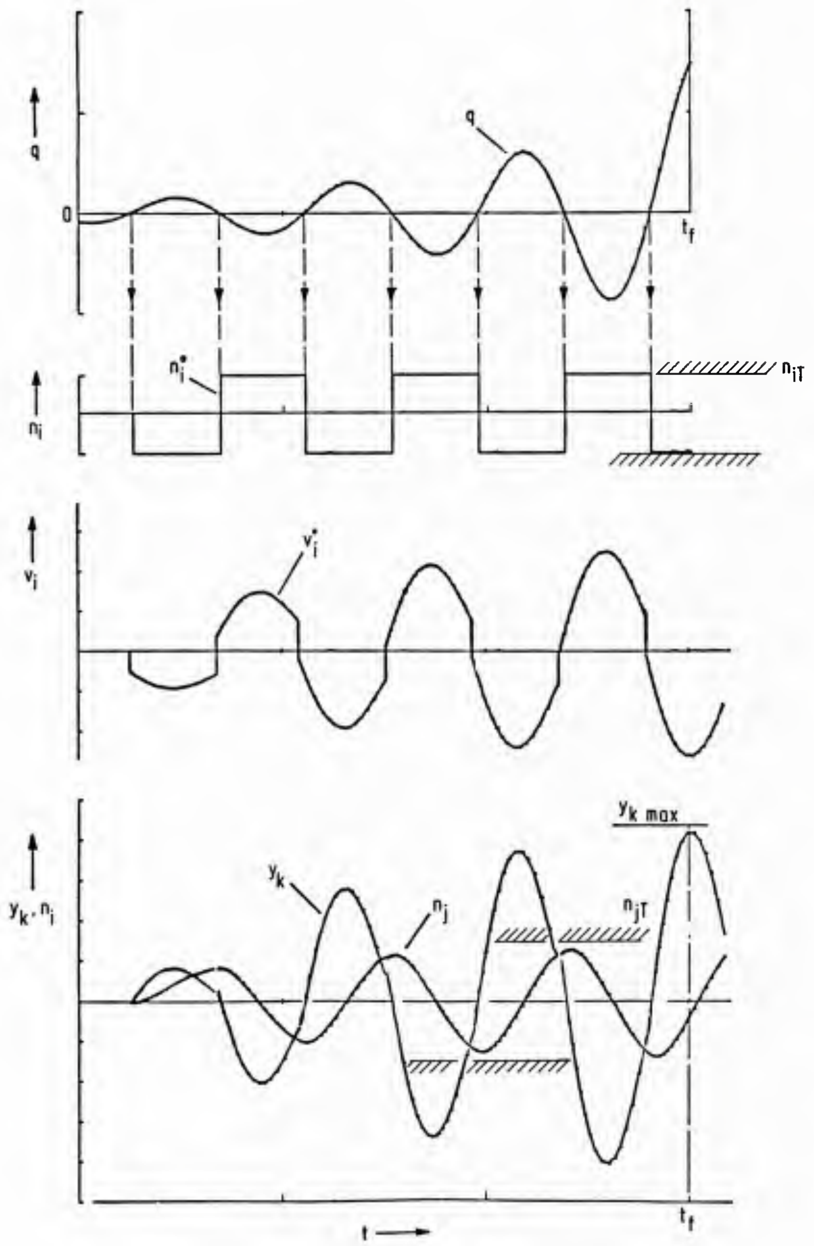


Fig. 3: Optimal time histories with limited input signal n_i

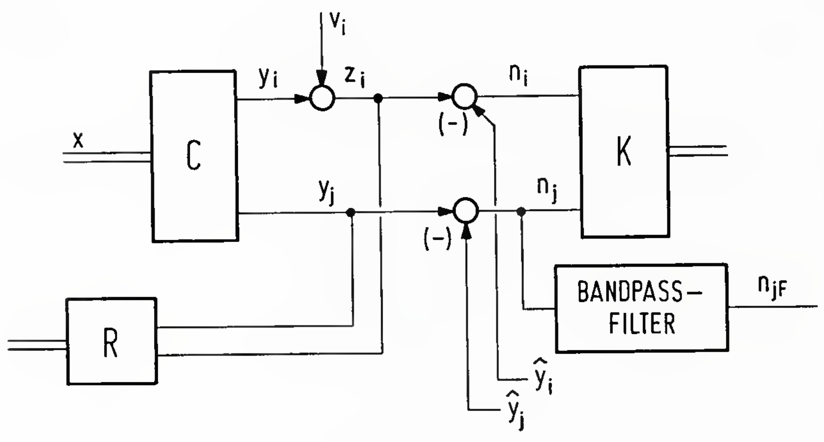


Fig. 4: Additional bandpassfilter

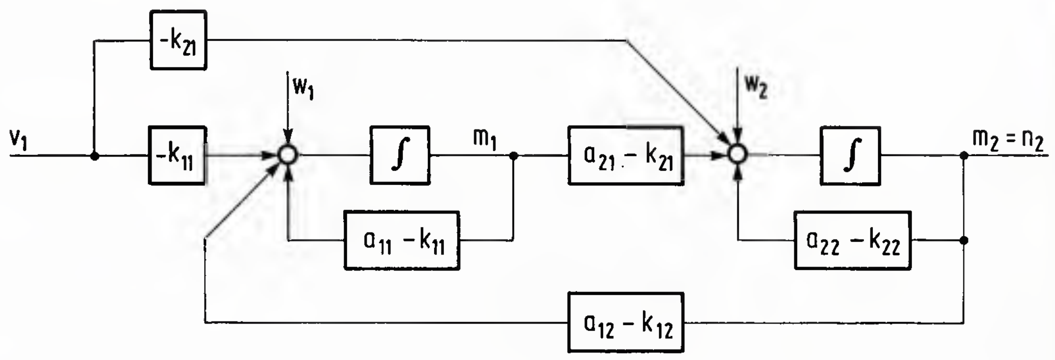


Fig. 5: Block diagram of a failure detection problem applied to a 2nd-order system

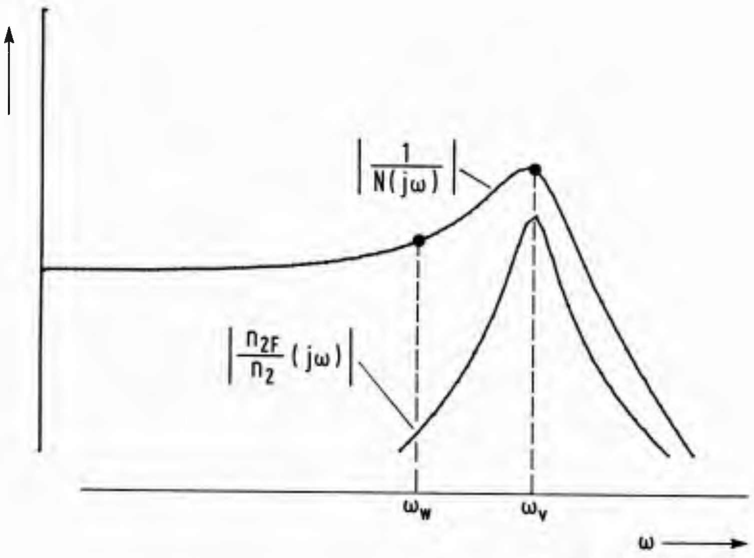
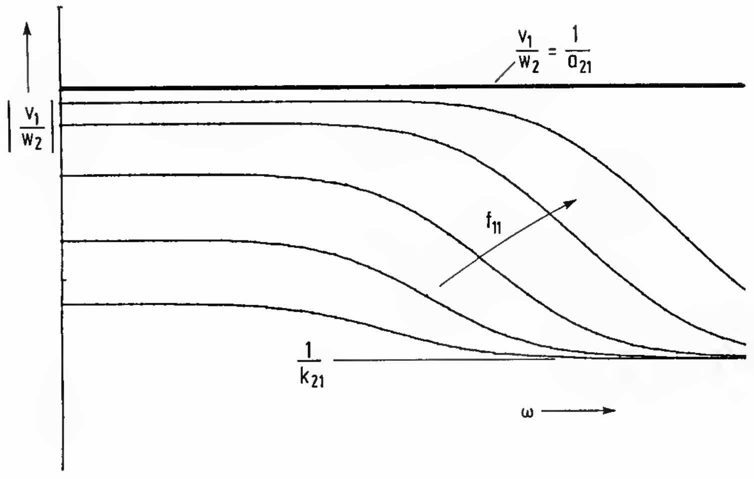
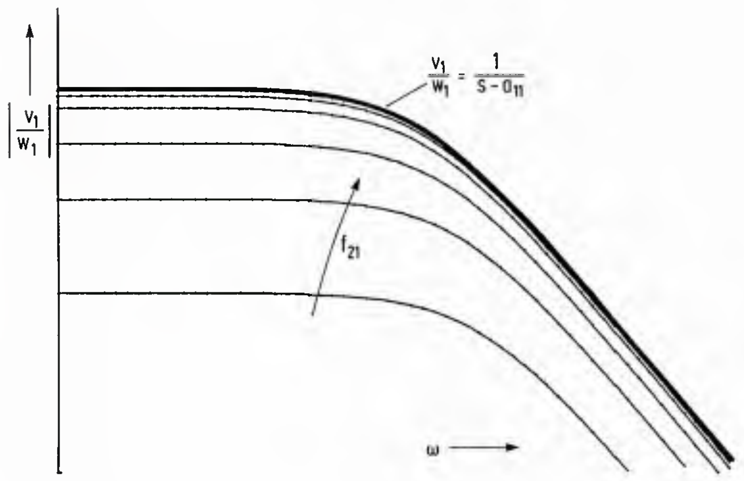


Fig. 6: Bode amplitude plots of the observer eigendynamic and of a bandpassfilter



a: Case $f_{11} >> f_{21}$



b: Case $f_{21} >> f_{11}$

Fig. 7: Bode amplitude plots of the sensorfailure-disturbance relations

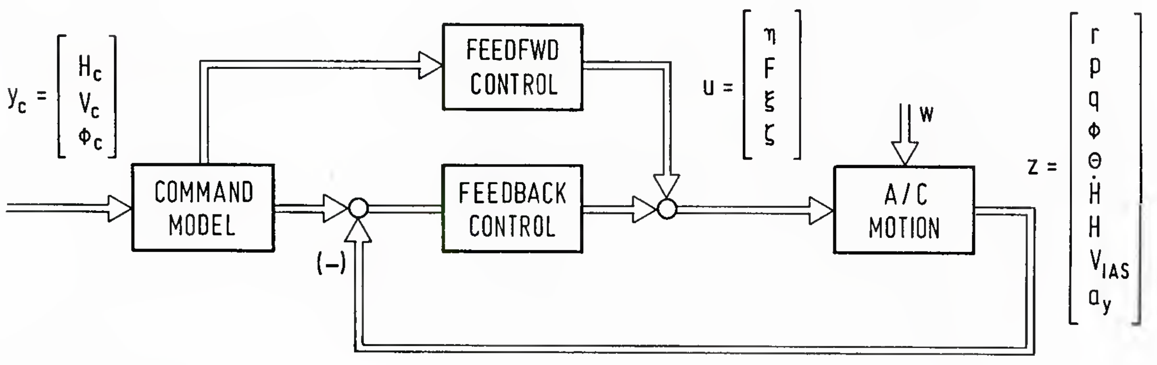
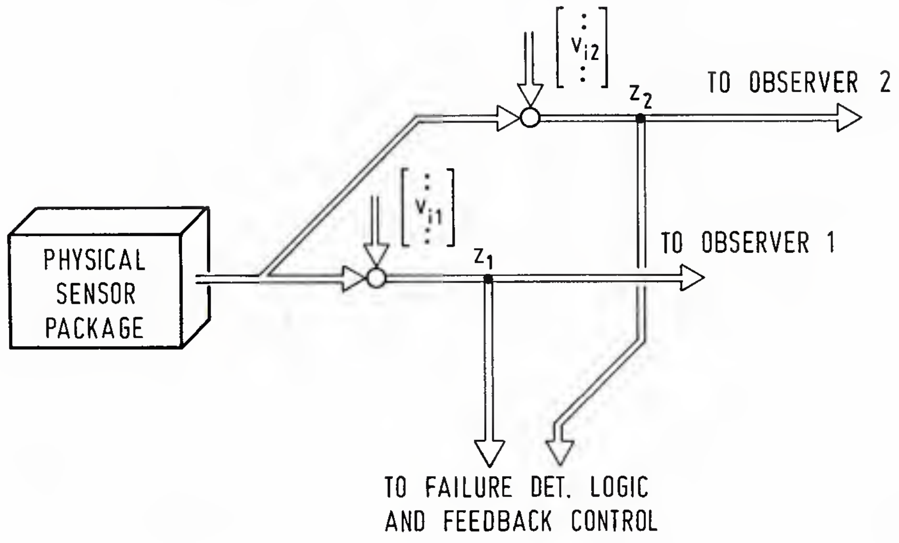

$y_c = \begin{bmatrix} H_c \\ V_c \\ \Phi_c \end{bmatrix}$
COMMAND MODEL
FEEDFWD CONTROL
FEEDBACK CONTROL
(–)
$u = \begin{bmatrix} \eta \\ F \\ \xi \\ \zeta \end{bmatrix}$
W
A/C MOTION
$z = \begin{bmatrix} r \\ p \\ q \\ \Phi \\ \Theta \\ \dot{H} \\ H \\ V_{IAS} \\ a_y \end{bmatrix}$

Fig. 8: Flight control system for a transport aircraft



v_{i1}: Additive signal in sensor i of SP 1

v_{i2}: Additive signal in sensor i of SP 2

Fig. 9: Software realization of two sensor packages

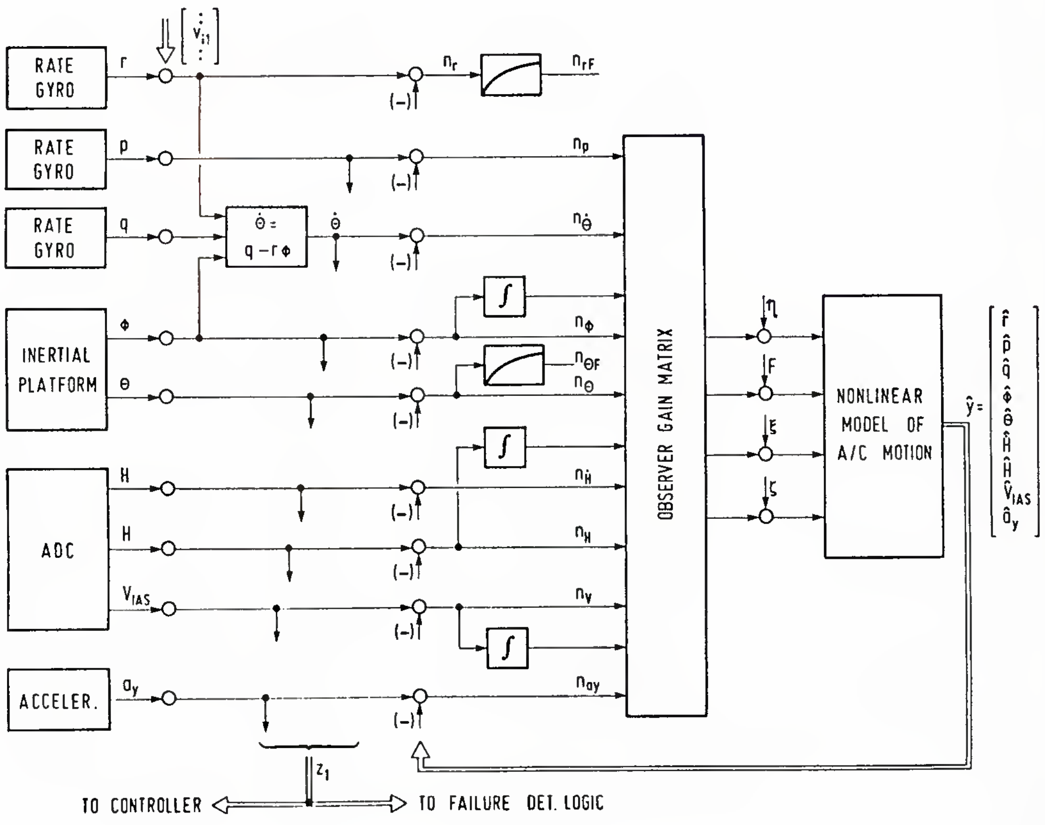


Fig. 10: Experimental sensor and observer arrangement

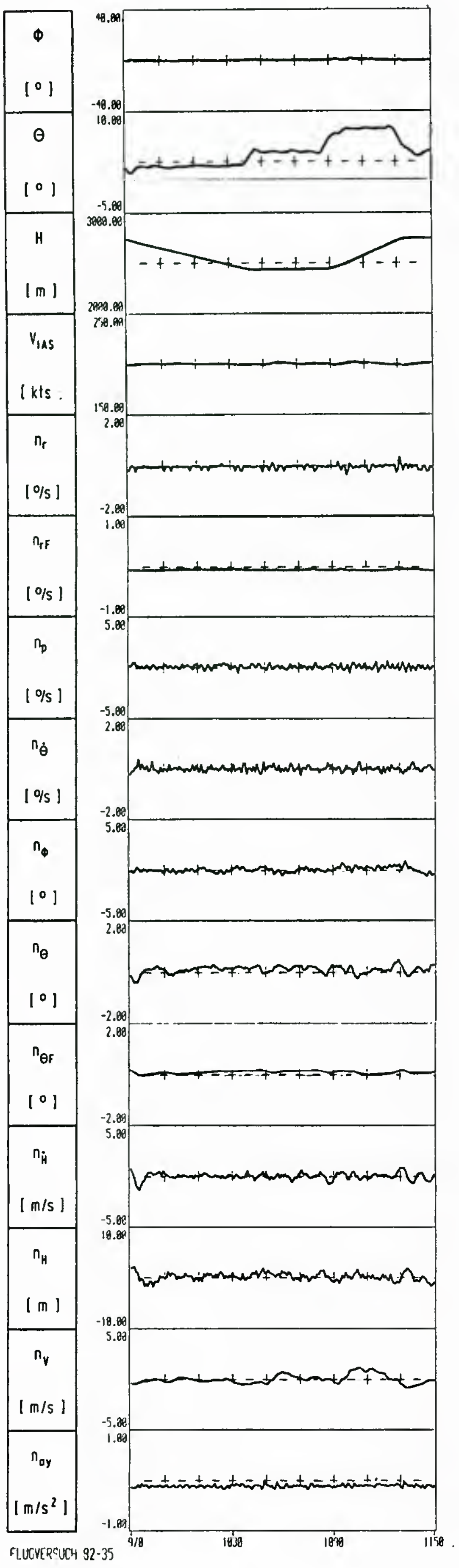


Fig. 11: Response to altitude commands

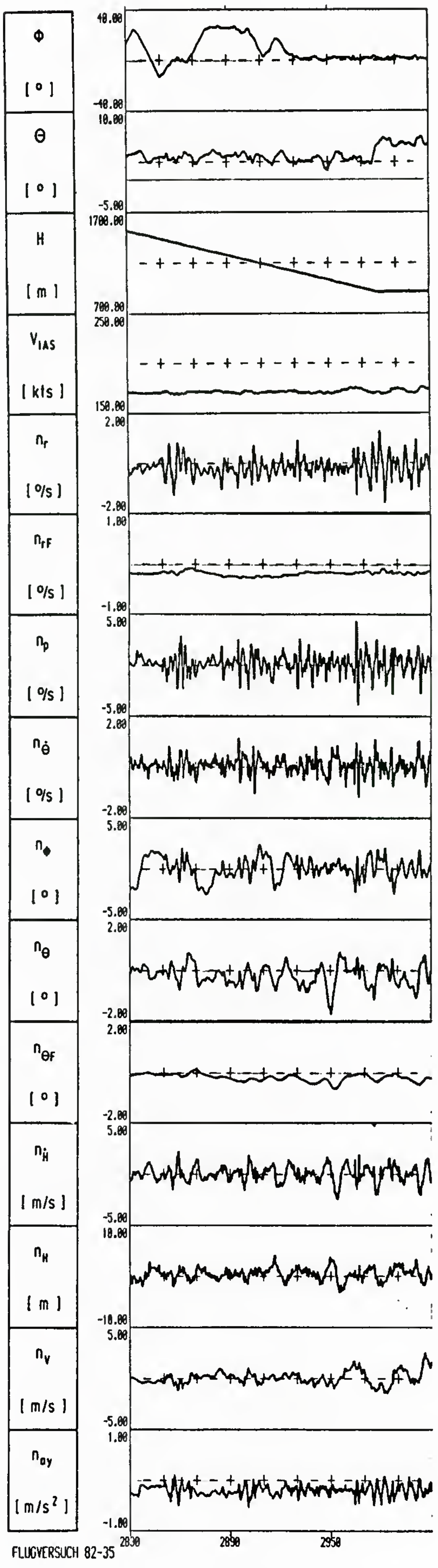


Fig. 12: Response to heavy turbulence during descent including bank attitude commands

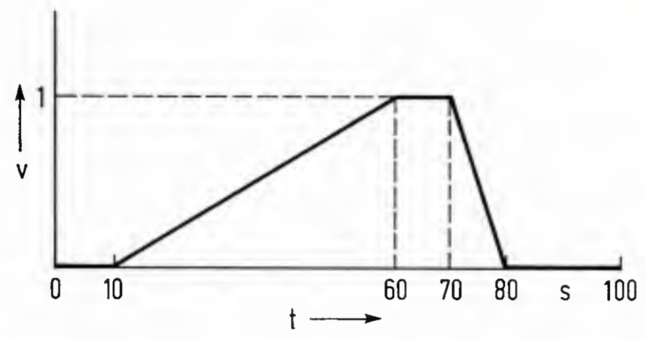
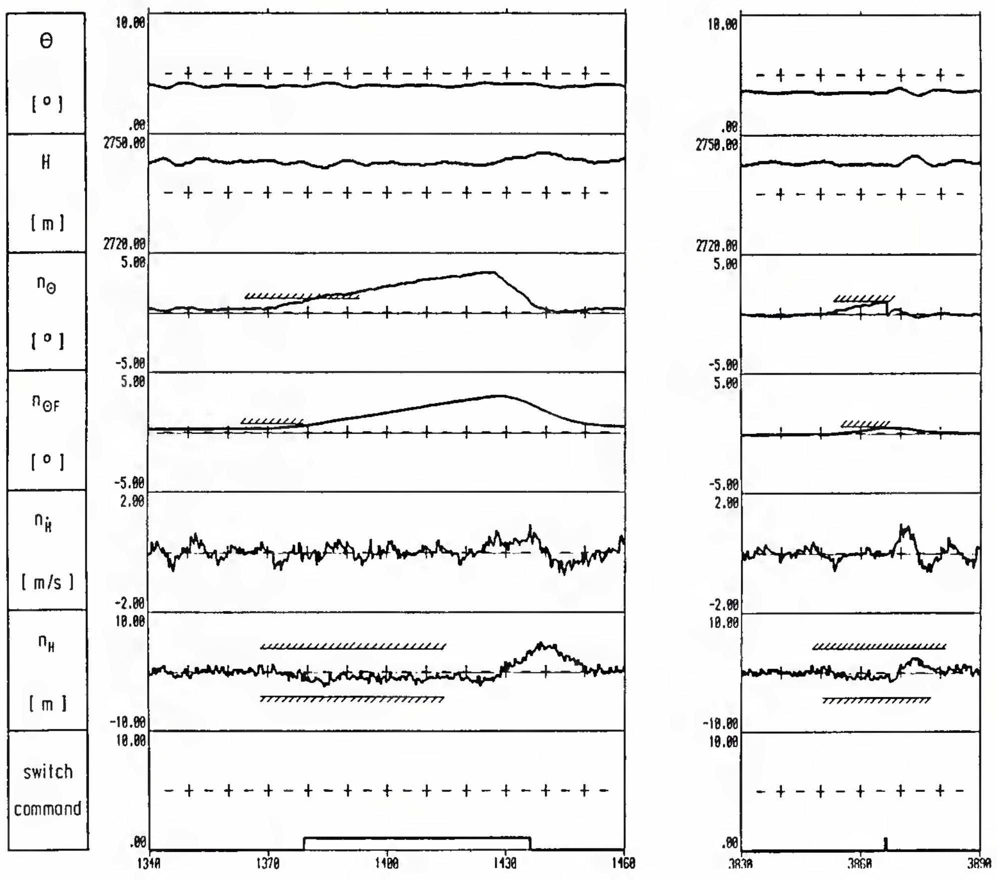


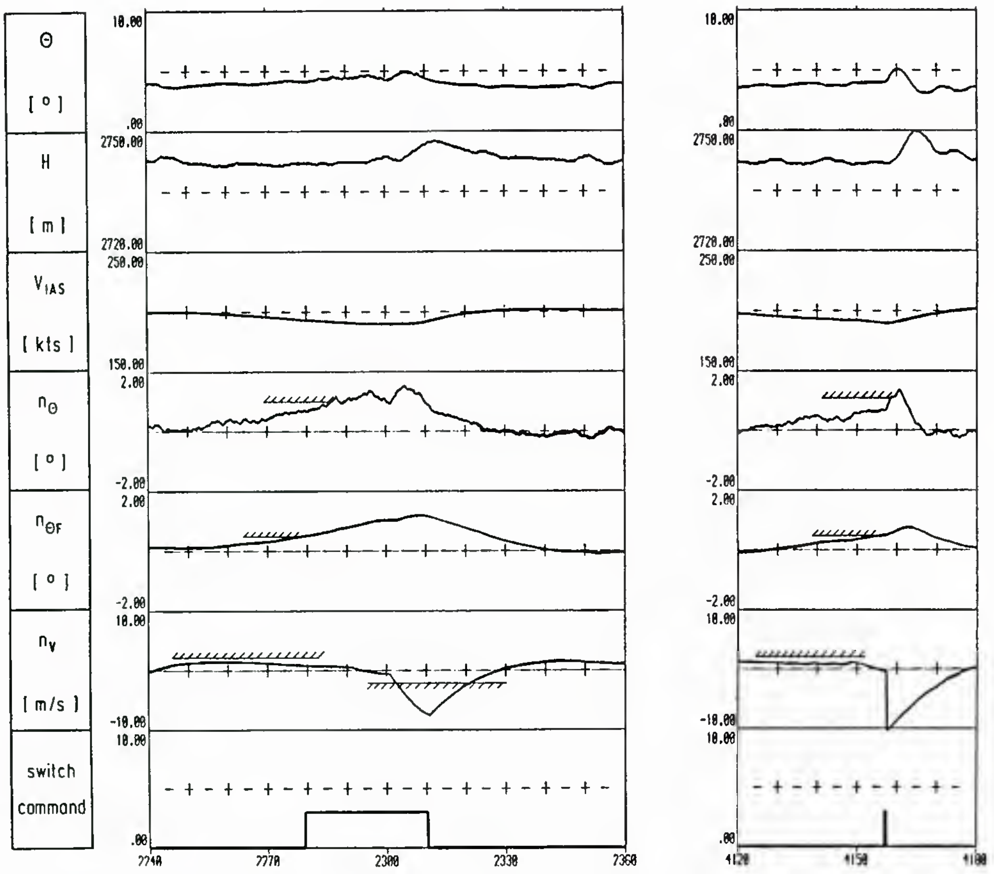
Fig. 13: Common sensor failure profile



a. without switching

b. with switching

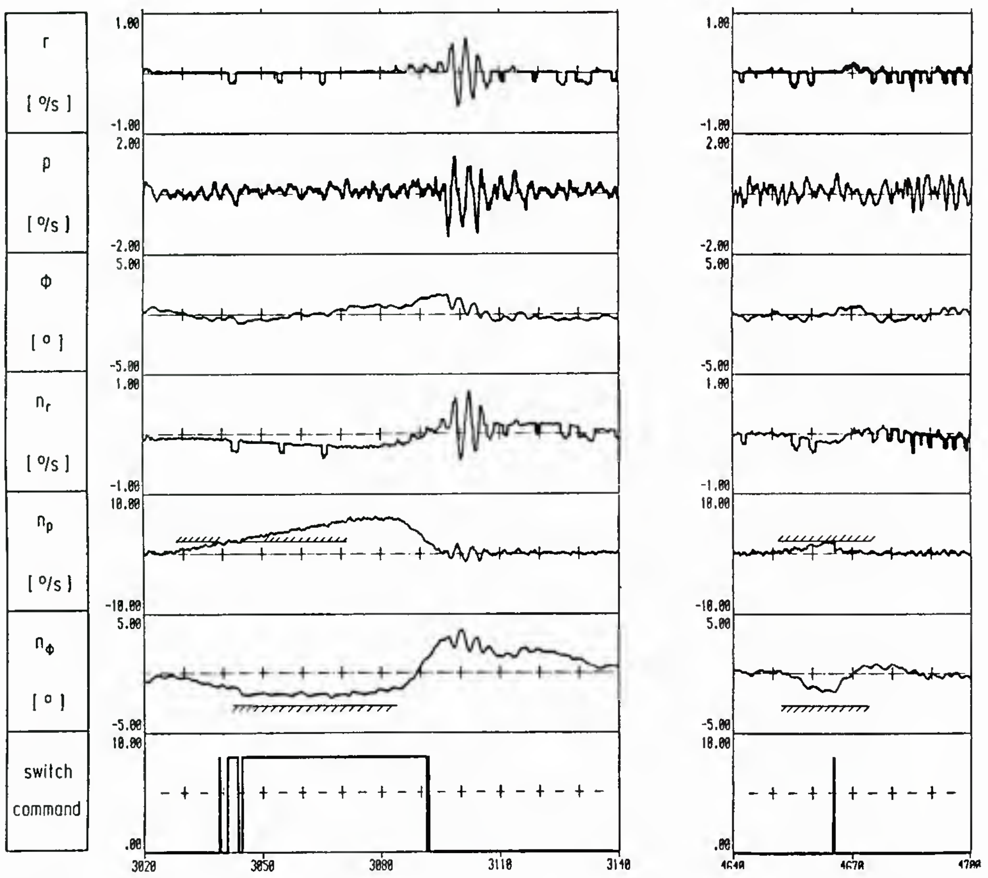
Fig. 14: Response to failures in the Θ-sensor



a: without switching

b: with switching

Fig. 15: Response to failures in the V_{IAS}-sensor



a: without switching

b: with switching

Fig. 16: Response to failures in the p-sensor

ASSESSMENT OF SOFTWARE QUALITY FOR THE AIRBUS A 310 AUTOMATIC PILOT

R. TROY
VERILOG
Le Péripole
3, Chemin du Pigeonnier de la Cépière

31081 TOULOUSE CEDEX
FRANCE

C. BALUTEAU
SFENA
B.P. 59

78140 VELIZY VILLACOUBLAY
FRANCE

ABSTRACT

As for the whole of the Automatic Flight Control System, the automatic pilot for the AIRBUS is a computer system with high criticality. In order to assess the quality of the software, a number of observations and measurements have been performed during the development cycle. This paper presents a general method of evaluation, the data collected, and some evaluation results.

The aims of this analysis are to assess the impact of development constraints on the quality of the product and to evaluate its operational reliability. The results show the conditions in which software may be adversely affected by modifications. In particular, they make it possible to appreciate the quality of the design, the effectiveness of the verification and validation processes, and the flexibility of the software. The last part gives a tentative example of operational reliability evaluation, taking into consideration the complexity of the functions and the mission profile.

INTRODUCTION

The Automatic Flight Control System for the AIRBUS A310 is a computer system with high criticality.

The constraints of Safety and Availability have made it necessary to use fault-tolerant techniques.

As a majority of the functions are performed by software, the developments and modifications are controlled by a production environment.

One important development constraints is the rapid and continuous change in specifications to which, owing to its very nature, the software is subject. In order to objectively appreciate the performance of the production facilities, the impact of the modification constraints and the quality of the software resulting an Observation and Measurement procedure was organized. This paper presents the assessments resulting from a first exploitation of these data. The aim of this preliminary analysis was to show that a judicious collection of data, complemented by an efficient evaluation method, can produce significant information which makes it possible to assess the phenomena affecting the quality of the product, such as regression in the course of modifications, the effectiveness of the verification and validation tests, etc ...

Following a presentation of the project in paragraph 1, the aims of the analysis are developed in paragraph 2. These aims concern the impact of development constraints and the evaluation of operational reliability. The evaluation method is then described in paragraph 3 ; it consists of three stages : Observation and Measurement, Statistical Analysis and Modelling.

The Observations and Measurements are presented in paragraph 4 where a distinction is made between the data characterizing the product, the development process and the utilization profile.

Some example of evaluations are given in paragraph 5 in order to illustrate how it is possible to fulfill the objectives set in paragraph 2.

1 - CONTEXT OF THE ANALYSIS

1.1. - AIRBUS A310 AUTOMATIC FLIGHT CONTROL SYSTEM

The System under review is an on-board, hardware and software system which contains six different types of computers and performs all the Automatic Flight Control System (AFCS) functions.

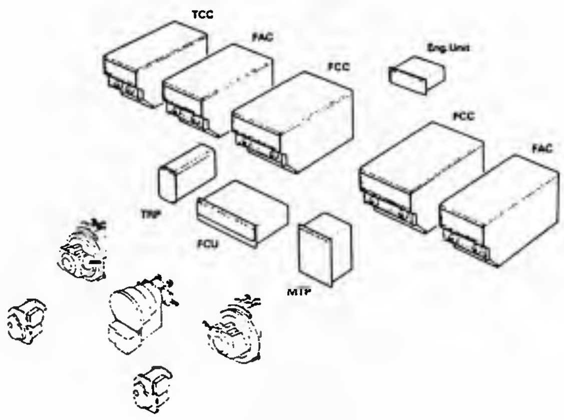


Figure I

Distribution of Main Functions :

- FLIGHT CONTROL COMPUTER (FCC)
  - automatic pilot and flight director with autoland function.
- FLIGHT AUGMENTATION COMPUTER (FAC)
  - pitch trim,
  - yaw damping,
  - flight envelope protection.
- THRUST CONTROL COMPUTER (TCC)
  - control of speed and maximum engine setting,
  - maximum engine setting computation.

These three computers have a built-in test and fault isolation system.

Fault-tolerant Architecture

The design of the system is governed by two important requirements : Safety and Availability.

These requirements make it necessary to have a fault-tolerant architecture consisting of

- two independent computing channels based on asynchronous processors : for the critical functions, each of the computing channels receives its own information from outside the computer ;
- hardware voters associated with comparators which detect any discrepancies between the computing channels.

On a global level, survival after a first failure is provided by a second computer identical to the first.

1.2. - SOFTWARE LIFE-CYCLE

The majority of the system functions are performed by software. In total, the programs represent 230 Kbytes written 60 % in high-level language (PLM 86) and 40 % in assembler language for the 8085 / 8086 microprocessors. The software development process is divided into eight phases : external specifications, preliminary design, detailed design, production, individual tests, integration tests, software verification, validation.

For each phase, there is a verification of the products resulting from it. The development of the first software version was accomplished in 10 months (from October 80 to September 81) by a team of 40 engineers and programming analysts. This phase was followed by an intensive modification cycle lasting 18 months. Commercial exploitation started in March 1983 with the airlines LUFTHANSA, SWISSAIR and KLM. At the present time, the system has accumulated a total of 6000 flight hours.

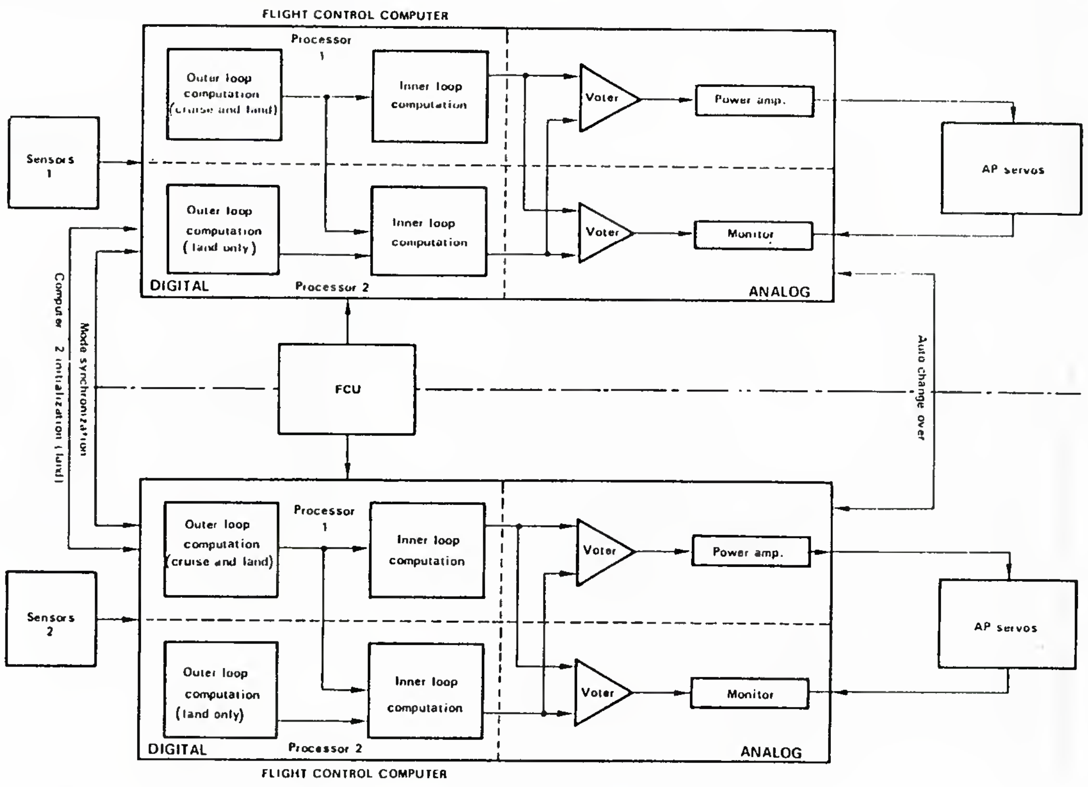


Figure II

Development Constraints

Not all the data required to establish the final specifications are available right at the start of an avionics software project (aircraft modelling for example). Consequently, the rapid and continous change of operational specifications represents an important feature of SFENA softwares. It involves a succession of modification sequences motivated both by specification changes called Development Change Requests (DCR) and by the correction of errors found during validation.

The following diagram represents the production of successive software versions. A distinction can be seen between the versions produced during integration, which are shown as I_n, and the versions produced as a result of modifications, which are show, as V_n ; n represents the number of the version.

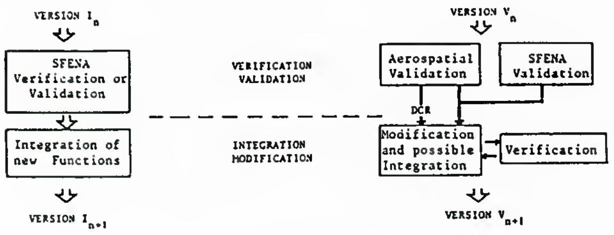


Figure III

while its aim is to improve the operational performance and quality of the software, special attention must be paid to the modification process as it may have the opposite effect to that desired. In particular, it may lead to a premature regression of software performances if the design principles and development facilities are not suitable for it.

Software Production Environment

The software production environment set up to meet the requirements mentioned above consists of processors developed by SFENA on INTEL systems. Without going into a detailed description, two of these processors have a particularly important role :

- the modification processor : MODIFY

  Program modifications are implemented from the source texts of the modules in the current version. In order to avoid direct handling of the source texts, modification items are constructed containing : the reasons for modifications, a limited number of instructions (add, delete, replace) applicable to the numbered source lines, and descriptive commentaries. The modification item represents the difference between two successible versions of an element. The new source text is generated by MODIFY.

- The software production processor : MODGEN

  This processor makes it possible to sequence all the operations required to obtain an executable code. During the integration of a software or the production of a new version, it automatically sequences the following operations :

  . call-up of basic processors (MODIFY, compilers, link editors, memory image constructors) ;

  . generation, for each operation, of an "error log" file ;

  . production on the console of an echo of the operations performed ;

  . computing and checking of the file signatures obtained at all levels ;

  . update of the software nomenclature.

2 - AIMS OF THE ANALYSIS

This analysis is aimed at two different objectives :

2.1. - ASSESSING THE IMPACT OF PRODUCTION CONSTRAINTS ON PRODUCT QUALITY '

As in any industrial field, customer/supplier relations are subject to the constraints of lead-times, incorporation of modifications, etc. It is of course impossible to assess the impact of these constraints in an absolute sense, as this would assume that we would be able to cancel them out in order to have a reference ; on the contrary, the declared aim of these assessments is to achieve higher performance in proportion to their intensity. In consequence, it is assumed that these assessments will make it possible to appreciate the degree to which the SFENA production facilities are suited to these constraints, thereby revealing the critical points in the development cycle.

2.2. - FORECASTING OPERATIONAL RELIABILITY

The static characteristics of softwares (complexity) and their behavior as observed during tests provide input data for the models which make it possible to evaluate and forecast the operational reliability. (MUS, 80), (LIT, 81), (CHE, 80). These models introduce objective elements for deciding to stop a test, defining a test policy, providing the resources required to achieve a reliability objective, etc ...

Because it is implemented in the industrial field, this analysis has the advantage that it permits a concrete evaluation of the applicability of such models.

3 - METHOD

Given the declared aims of the analysis, the problem to be solved is to establish :

- the variables which give information on the phenomena to be studied ;

- the laws governing the development of these variables.

The solution to this problem is of the type which must be based on a procedure containing the following main stages : Observation, Statistical Analysis and Modelling.

- Observation makes it possible to acquire the necessary data specific to the analysis.
- The Statistical Analysis is on two levels :
  . the descriptive analysis which reveals the observable phenomena and makes it possible to determine the variables which give information ;
  . the exploratory analysis which compares different phenomena in order to suggest and establish the relationship between cause and effect.
- The Modelling establishes the laws, either empirical or theoretical depending on the outcome of the exploratory analysis, which govern the development of the variables (or phenomena) and provide the means of appreciating the quality of the models produced.

The main advantage of developing this method, which is described in detail in (ROM, 82), lies in the fact that each of the three stages corresponds to a type of decision that can be taken in the light of the results obtained :

- At the end of the first stage, we are in possession of the data believed to be necessary for the analysis, but we know nothing about the content of the data collected ; the decisions taken are of a subjective nature.
- After the analysis stage, the phenomenology of the failures is percieved and understood ; the results obtained make it possible to take objective decisions.
- The third stage makes it possible to predict future behavior and take objective decisions while appreciating the risk involved.

It is important to emphasize the fundamental role played by Observation in order to obtain satisfactory results at the end of the Modelling. The two characteristics which affect the quality of the measurements are :

- the definition of the observations,
- the dependability of the data collection process.

Paragraph 4 describes how the second characteristic has been satisfied. The first can only be approached in a system situation where, starting off with correct measurements, the method described previously is applied in order to bring out the first interesting phenomena and to define the measurements, before performing new observations which allow progress to be made in the understanding of the phenomena.

4 - OBSERVATIONS AND MEASUREMENTS

The results given in this paragraph concern the FCC software which performs the automatic pilot functions (cf. Figure I). This software consists of 56,000 instructions in source language, taking up 120 Kbytes of program and executed on four microprocessors.

The Observations and Measurements successively concern the product, the development process and the utilization.

4.1. - PRODUCT MEASUREMENTS

The characteristics of the product have been analyzed on two levels : the interconnection structure for the various functions and the complexity of each function.

4.1.1. - Interconnection Structure

Figure IV shows the sequencing operations that are possible in a cycle of 150 ms for the functions providing the main cruise and landing modes of the automatic pilot :

1 :	Initialization on power-up
2 :	Real-time monitor
3 :	Decoding of discrete data (pilot + environment)
4 :	Mode engagement
19 :	Mode disengagement
5 :	Go-around
6 :	Acquisition of selected heading
7 :	Yaw loop
8 :	Feedback
9 :	End
12 :	Vertical speed or altitude hold
13 :	Altitude Acquisition
14 :	Glideslope beam capture and tracking
15 :	Level capture and selected speed
16 :	VOR navigation or tracking
17 :	Localizer beam capture and tracking
18 :	Heading hold

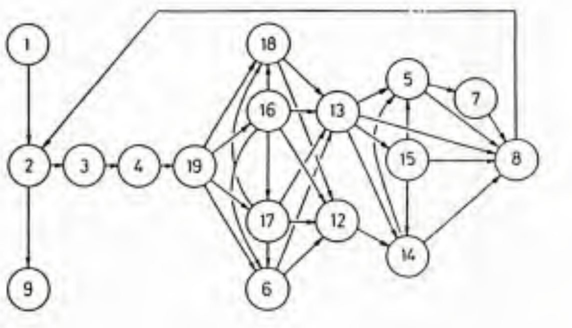


Figure IV

MEASUREMENT		MEANING	ACCEPTABLE RANGE	CALCULATED VALUE
Hierarchical complexity	CH	Mean number of modules per level	$]0, 10]$	1.14
Structural complexity	CS	Density of links between modules	$]0, 3]$	1.94
Accessibility of modules	A	Diversity of access to modules	$[0.01, 1]$	0.125
Testability of paths	TC	Effort required to run through a path on the graph during the test	$[10^{-3}, 1]$	$[0.002, 0.25]$
Testability of system	TS	Effort required to test the system	$[10^{-6}, 1]$	3.10^{-6}
Entropy	E	Disorder in inter-module relations	$[0, 50]$	6.63

Table I

Table I gives the results of the complexity measurements of this graph according to the measures developed by Mohanti (MOH, 79).

By analysing this table, it is possible to distinguish

- the highly satisfactory results : the levels are simple (CH), the functions are readily accessible (A), the paths are easy to test (TC) and the system is well organized (E) ; these results indicate that the software design has been well done ;

- the less favorable results concerning the difficulty in testing the system (TS) and the appreciable complexity of the relationships between functions (CS), which indicate that the application is, by its very nature, difficult.

4.1.2.- Complexity of Functions

The functions involved are characterized by numerous internal logic conditions and a high global flow of data. As a result, the complexity of each function has been established by taking into consideration

- the complexity of the checking structure measured by the Entropy (EM) and the Checking Density (DC) ;

- the complexity of the data flow measured by the number of global inputs (NE) weighted by the percentage of instructions for accessing the global variables (P) ;

according to the formula :

$$C = 0.5 \times \left\{ \left(\frac{EM}{EM_{limite}} \times \frac{DC}{DC_{limite}}\right) + \left(\frac{NE}{NE_{max}} \times P\right) \right\}$$

The results obtained are given on the following table :

MODULES	DC/0.5	EM/15	NE/NE max	P	C
1	0.95	0.37	2.3	0.1	0.29
2	0.95	0.35		0.	0.33
3	0.75	0.53	2.2	0.42	0.66
4	1.02	1.66	9.58		1
5	0.8	1	2.5	0.21	0.675
6	1	0.17	1	0.19	0.18
7	0.81	0.85	4.16	0.16	0.673
12	0.875	0.4	2.83	0.5	0.83
13	0.8	0.33	0.67	0.185	0.195
14	0.952	0.53	3.33	0.31	0.77
15	0.77	0.27	2.5	0.28	0.45
16	0.78	0.4	2	0.33	0.49
17	0.875	0.34	2	0.31	0.45
18	0.93	0.38	2.33	0.378	0.62
19	1.01	1.8	3.92		1

Table II

4.2. - OBSERVING THE DEVELOPMENT PROCESS

The development has only been observed from the Integration. In order to be effective, the collection of data must be uniform for all items of equipment, whithout loss of information and with no subjectivity. In order to meet these objectives, procedures were set up during the development of the project. Today, they are automated to a great extent.

Data "sensors" are placed both in the Validation process and in the Modification process (cf. Figure III). These observations must be organized in such a way as to respect normal working practices and to integrate themselves into the existing structures.

Whenever a deficiency is discovered in the course of a Validation, a sheet provided for the purpose is filled in immediately. Initially, this sheet gives the identification of the product and its version, together with a description of the deficiency. This sheet is numbered and distributed to the team in question. After investigation, the Software Manager completes the sheet by describing :

- the fault in the program,

- the correction made,

- the class of the anomaly.

The sheets are filed at project level for each subsystem. The anomalies are then taken into consideration in the Modification process, with no possibility of losing or omitting information.

By using the "reasons for modification" directives of the MODIFY processor, the references of the anomalies and their encoded classes are acquired at the level of the creation of the change items for the modules in question. Having been filled in, the change items are submitted to the MODGEN production processor.

Then, a first tool is used to collect and sort all the reasons for modification and to point out any syntax errors that may have crept in during acquisition. After correction, the production stages continue until the executable code is produced.

Table III shows :

- for each software version : the number of errors detected in the Verification / Validation process (cf. Figure III) together with their classification according to the phase of origin and the type of error.

  The classes or errors are defined as follows :

  A : design (program architecture, breakdown into modules) .

  B : arithmetic and logic (false control law, false logic equation, incorrect conversion, etc ...).

  C : control logic (loop error, branch error, false initialization, etc ...).

  F : interface (incorrect encoding / decoding of input / outputs, etc ...

  Z : persistent (error surviving after corrections).

- to characterize the Verification / Validation process : the number of test hours and the consequent effort calculated on the basis of a new input applied to the software every 150 ms.

In commercial service, the SFENA customer support department has access to the same type of anomaly sheets, enabling the deficiencies in the systems to be collected together. Their origins (hardware, software, environment) are determined in laboratory investigations. The processing of these sheets is the same as for the sheets obtained by the validation process.

The GENSTA processor installed recently uses the activity reports from MODGEN. GENSTA collects the data specific to the modification of each module. The amount of data collected is aimed at correlating the parameters affecting the quality of a software and at forescasting its behavior on the basis of its past history. This processor is activated by MODGEN in an entirely user-transparent manner. GENSTA gathers and classifies the following data :

- for each module :
  - the number of modified source lines ;
  - the number of tests performed over the whole production cycle ;
  - the working time of the MODIFY modification processor.

VERSION NUMBER	TEST HOURS	TEST EFFORT	NUMBER OF ERRORS	PHASE OF ORIGIN			TYPE						DCR NUMBER	NUMBER OF MODULES MODIFIED
				ANALYSIS	CODING	MODIFI-CATION	A	B	C	F	Z	OTHER		
I0	48	1.15×10^6	11	4	7	0	0	4	0	0	3	3	-	-
I1	96	2.30×10^6	31	3	27	1	0	7	4	15	0	5	-	-
I2	96	2.30×10^6	18	3	14	1 •	1	8	1	4	1	3	-	-
I3	96	2.30×10^6	17	5	12	0	0	11	2	3	1	0	-	-
I4	96	2.30×10^6	23	10	13	0	3	12	0	6	1	1	-	-
V1	200	4.8×10^6	50	20	15	15	9	6	4	9	16	6	13	36
V2	320	7.68×10^6	73	46	7	20	5	19	4	8	21	16	13	47
V3	190	4.56×10^6	37	7	23	7	1	3	9	9	2	13	12	24
V4	150	3.6×10^6	30	12	18	0	1	11	3	4	5	6	16	46
V5	220	5.28×10^6	47	20	9	• 18	3	5	12	7	7	13	19	72
V6	80	1.92×10^6	13	8	3	2	0	5	3	1	0	4	9	30
TOTAUX	1592	$38,2 \times 10^6$	350	138	148	64	19	49	35	38	51	58	82	255

Table III

The software for an item of equipment generally uses several softwares which are produced separately and executed on different processors. From the files produced in the course of MODGEN, the tool "ATTAC8" collects and classifies all the modifications made on the various softwares related to one item of equipment, giving for each reason for modification the list of the elements affected in each software. This list is a component of the documentation associated with the equipment. It also makes it possible to give a complete breakdown of all the changes undergone by the softwares during their life cycle.

- for each equipment processor :
  - . the total duration of MODIFY ;
  - . the total working time on the development system ;
  - . the amount of work submitted ;
  - . the number of modules modified ;
  - . the total number of new source lines ;
  - . the average modification rate ;
  - . the average size of a modification.

As an example, Table IV gives a synthesis of these data for the last four versions.

5 - EVALUATIONS

OBJECTIVE 1 - Impact of Development Constraints

The data given in tables III and IV make it possible to perform the following analyses :

5.1. - DESCRIPTIVE ANALYSIS

With reference to table III, this paragraph presents the characteristics and performances of the Verification /Validation and Integration / Modification processes.

VERSION	MODULES MODIFIED	NEW MODULES	DIRECTIVES	NEW LINES	AVERAGE DIRECTIVE LENGTH	MODIFY USE TIME	JOBS PERFORMED	SYSTEM TIME	LINES MODIFIED	NUMBER OF TESTS	NUMBER OF IMPROVE-MENTS
Vo3	74	10	1195	3119	3	14^{h}41	795	48^{h}56	27441	412	33
Vo4	102	16	1363	4446	3	10^{h}59	648	31^{h}36	32177	358	28
Vo5	103	1	1365	4386	3	19^{h}20	1091	51^{h}07	29458	570	23
Vo6	70	4	1156	3168	3	6^{h}17	404	24^{h}05	23261	174	

Table IV

4.3. - OBSERVING THE UTILISATION

The FCC software has accumulated more than 6000 flight hours. Of the observed anomalies that have given rise to report sheets, none could be ascribed to the software.

Recent studies have shown the impact of the software utilization range on operational reliability (IEY, 82), (ROM, 82), (BUT, 80), (ROS, 82). Account with this result the frequency with which each of the FCC functions is activated has been measured on different routes. Table V shows these frequencies for two routes : TOULOUSE - LONDON and LONDON - AMSTERDAM.

TRANSITION	TOULOUSE - LONDON	LONDON - AMSTERDAM
1 - 2	1	1
2 - 3	(n-1)/n	(n-1)/n
2 - 9	1/n	1/n
3 - 4	1	1
4 - 19	1	1
19 - 16	0.8	0
19 - 17	0.14	0.25
19 - 6	0.01	0.69
19 - 18	0.04	0.04
16 - 18	0.05	0
16 - 6	0.05	0
16 - 12	0.51	0
16 - 13	0.36	0
16 - 17	0	0
17 - 6	0.16	0.45
17 - 18	0	0
17 - 13	0.83	0.54
17 - 12	0	0
6 - 12	0.14	0.53

TRANSITION	TOULOUSE - LONDON	LONDON - AMSTERDAM
6 - 13	0.85	0.46
18 - 12	0.5	0.66
18 - 13	0.5	0.33
12 - 14	0	0
12 - 13	1	1
12 - 8	0	0
13 - 14	0.14	0.13
13 - 15	0.27	0.32
13 - 5	0.15	0.04
13 - 8	0.43	0.48
14 - 5	0.16	1
14 - 8	0.83	0
15 - 14	0	0.26
15 - 5	0	0
15 - 8	1	0.73
5 - 7	1	1
5 - 8	0	0
7 - 8	1	1
8 - 2	1	1

Table V

5.1.1.- Verification / Validation

Characteristics

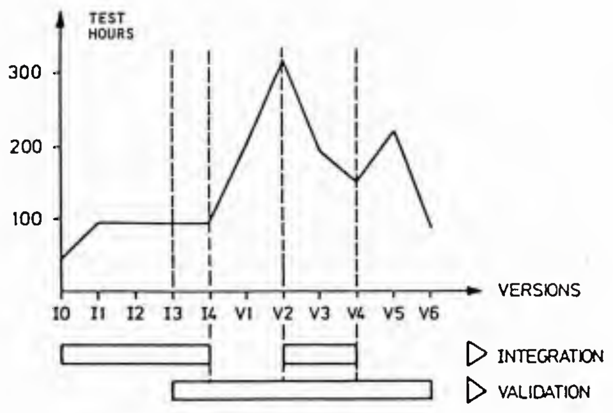


Figure V

The following features may be noted

- the start of the closed-loop validation tooks place as from I3 ;
- the integration of a new landing function after V2 ;
- the increase in the number of validation hours as soon as the airframe manufacturer enters into the validation process.

Performances

Distribution of detected error rates according to their phase of origin.

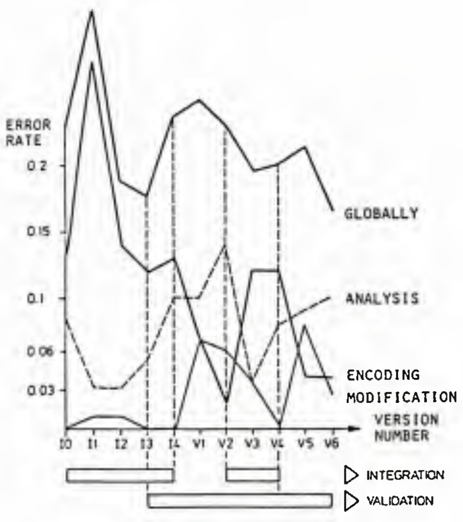


Figure VI

This serie of curves demonstrates

- the predominant detection of errors committed during coding, in the integration test ;
- the efficiency of the closed-loop validation process for finding the errors resulting from the analysis phase ;
- the non-negligible extent of the errors committed in the course of modifications.

The observations confirm the known results (ALB, 82), (LIP, 79), (MOT, 77), particularly concerning :

- the errors resulting from the design, which are revealed more effectively by a functional test than by a structural test ;
- the errors resulting from the coding, which are the most frequent and the easiest to detect.

Distribution of detected error rates accroding to their type

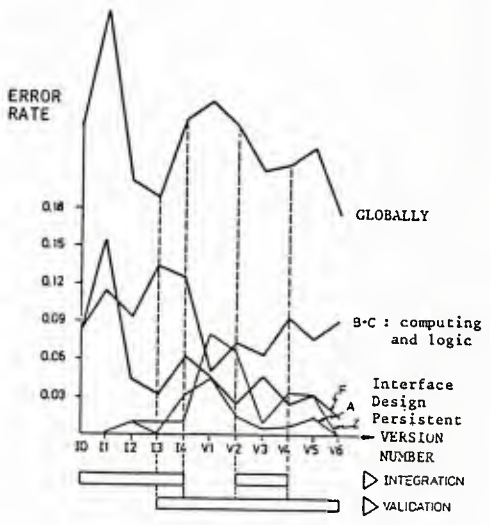


Figure VII

These curves show some significant phenomena which confirm and complete the previous observations

- the interface errors (F) are revealed during the integration tests, which is normal because it is during these tests that the computer interface is considered for the first time ;
- the computing and logic errors (B and C) are the most frequent and are best detected during the verification tests ;
- the design errors (A) are best detected by the closed-loop tests.

The "persistent" feature (Z) seems to be strongly affected by the nature of the test : it appears in fact that the closed-loop test detects errors which are more difficult to isolate (linked to the design).

These analyses make it possible to draw the following initial conclusions

- the integration phase has achieved its objectives : the interface errors and design errors have been revealed to a high degree ;
- the closed-loop test is effective : the analysis and design errors are preponderant here, which was not the case with the integration.

5.1.2.- Integration / Modification

Characteristics

The workload to be taken into consideration for each change, represented by the number of errors and the number of DCR's (development change requests), is distributed as follows :

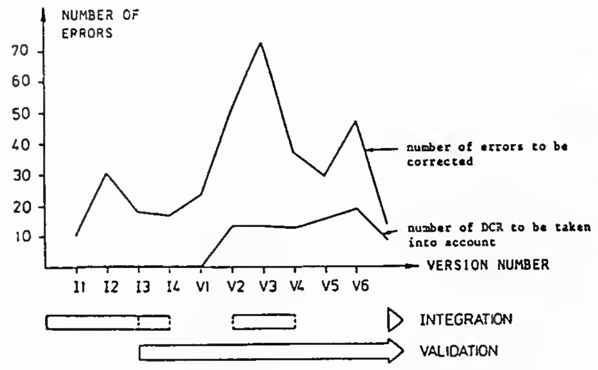


Figure VIII

Performances

The measurements performed by ATTAC8 make it possible, in particular, to establish the following histogram which provide information on the flexibility of the software.

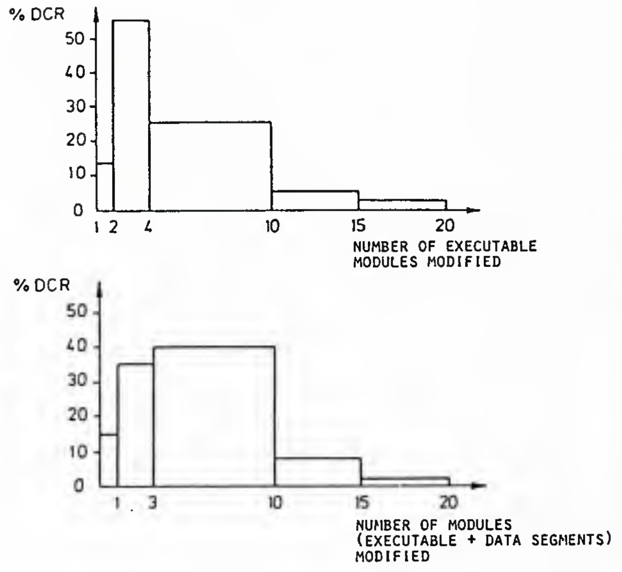


Figure IX

In order to limit the effects of the DCR's, which are usually modifications of the gains in the servo-systems, i.e. constants, the design of the programs has been carried out by separating the data from the executable code. It seems that this principle is insufficient, as it can clearly be seen that the majority of DCR's affect 3 to 10 modules.

Figure X shows the number of modules modified at each change of version.

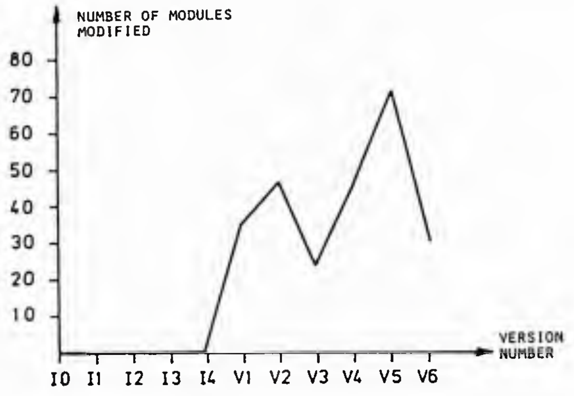


Figure X

5.2. - EXPLORATORY ANALYSIS

Specification changes (DCR's) represent an important control production constraint, and it is advisable to control their effects. As an illustration, a parallel has been made between the observed error rates, the number of DCR's and the number of modules modified.

A first examination (cf. Figure XI) shows an inverse trend between :

- the number of DCR's and the number of modified modules, which increase ;
- the rate of observed errors, which decreases.

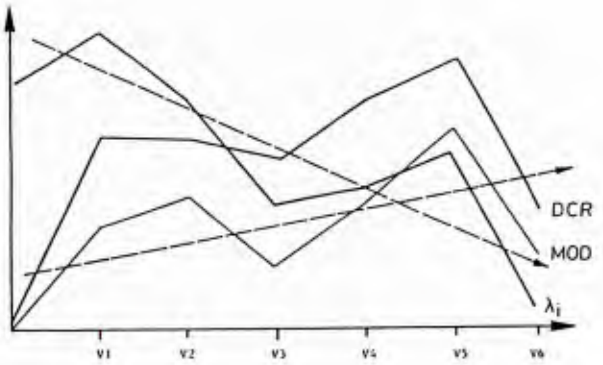


Figure XI

	λ_i	DCR_i	MOD_i	$DCR_i - DCR_{i-1}$	$MOD_i - MOD_{i-1}$	λ_i/λ_{i-1}	$\sum_1^i DCR$
λ_i	1.00						
DCR_i	0.39	1.00					
MOD_i	0.30	0.87	1.00				
$DCR_i - DCR_{i-1}$	0.59	0.50	0.25	1.00			
$MOD_i - MOD_{i-1}$	0.53	0.74	0.62	0.90	1.00		
$\lambda_i / \lambda_{i-1}$	0.76	0.81	0.63	0.90	0.97	1.00	
$\sum_1^i DCR$	-0.50	0.09	0.28	-0.68	-0.46	-0.33	1.00

λ_i represents the detected error rate during the validation of version i.

DCR_i represents the number of DCR's applied to version i.

MOD_i represents the number of module in version i that have been modified.

It can be seen in particular that :

- there is a highly negative correlation between λ_i and $\sum_1^i$ DCR, which expresses the trend inversion observed ;
- there are strong correlations between $\lambda_i / \lambda_{i-1}$ and $DCR_i - DCR_{i-1}$ or $MOD_i - MOD_{i-1}$.

5.3. - MODELLING

A multilinear regression makes it possible to express the cause-effect relationship between the numbers of DCR's and modified modules for two successive versions and the increase in the error rate.

The model obtained is as follows :

$$\frac{\lambda_i}{\lambda_{i-1}} = 0.3508 \times 10^{-2} \times (MOD_i - MOD_{i-1}) + 10^{-2} \times (DCR_i - DCR_{i-1}) + 0.9289$$

$$\rho = 0.97$$

It is interesting to exploit this model in order to study the conditions in which DCR's have a negative impact on the state of the version produced, i.e. in what conditions the software regresses. This is achieved by performing the equation :

$$MOD_i - MOD_{i-1} = \alpha\,(DCR_i - DCR_{i-1})$$

and studying the change in $\lambda_i / \lambda_{i-1}$ for various assumptions for α.

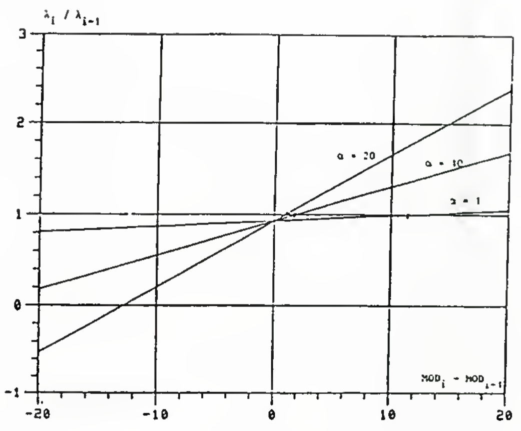


Figure XII

It can be seen that

- the state of the new version is not as good as the state of the previous version :
  - for two modifications if they each affect ten or so modules,
  - for twelve or so DCR's if only one module on average is modified per DCR.

This illustration demonstrates that the future state of the software depends to a great extent on concerted action : the intensity of the modifications and the flexibility of the software.

As the average number of DCR's is 13.6 per version, and the average number of modified modules is 3, it can be deduced from this that flexibility is a basic quality criterion for this type of software. In order to improve it, it appears necessary to anticipate the type of specification changes as early as the design stage.

<u>OBJECTIVE 2</u> - Operational Reliability Forecast

This analysis is based on

- the data provided in paragraphs 4.2 and 4.3 concerning the software characteristics and the utilization ;
- the results published in the literature on the relationship between the operational reliability of a program and the following characteristics :
  - code complexity (CHE, 80), (ALB, 82) ;
  - utilization (CHE, 80), (IYE, 82), (ROM, 82), (NOU, 82), (ROS, 82).

The presentation is limited to the resulting model.

In accordance with the schematic representation given on Figure IV, the software under study is seen as a network of independent functions that are created and tested separately.

The presentation is limited to the resulting model.

In accordance with the schematic representation given on Figure IV, the software under study is seen as a network of independent functions that are created and tested separately.

The reliability of a function is defined as the probability that it will correctly perform the service expected of it, i.e. producing the correct outputs and correctly transferring the control to the next function.

When the software is activated, a sequence of functions is executed. The result depends on this sequence and on the reliability of each function.

Assumption 1 : the reliabilities of the functions are independent.

The reliability of a function is modelled by a law which takes into account its complexity C_i, in accordance with the definition given in paragraphs 4.1.2. and the debugging word n_{pi} that has been applied to it according to the formula :

$$R_i = \frac{(1-C_i)\ (N_{pi} + 1)}{(1-C_i)\ N_{pi} + 1}$$

Assumption 2 : the transfer of control between functions takes place according to a Markovian process.

The transition to a state on the graph is a probability which depends only on the present state. It is considered that the probabilities of transition are constants which entirely characterize a given utilization profile. As an example, the transition frequencies given in table V characterize two distinct utilizations of the same software.

Reliability Model

The network of functions is represented by an oriented graph, each node of which is a function and each arc a possible transition. Each node is associated with a reliability R_i, and each arc with a probability P_{ij} which expresses the probability of a transfer of control between i and j.

Two absorbent states C and F have been added to represent the correct termination and the failure respectively.

A transition to F has been added to each node i with the probability $1 - R_i$ to represent the occurrence of a fault during the execution of i.

A transition to C has been added at the terminal function n with the probability R_n to represent the correct execution fo the output function.

The probability of transition P_{ij} between two functions i and j is weighted by R_i to represent the faultless execution of function i.

In consequence, the reliability of the software is the probability of reaching state C from the initial state.

Figure XIII represents the Markov chain modelling the FCC software under study.

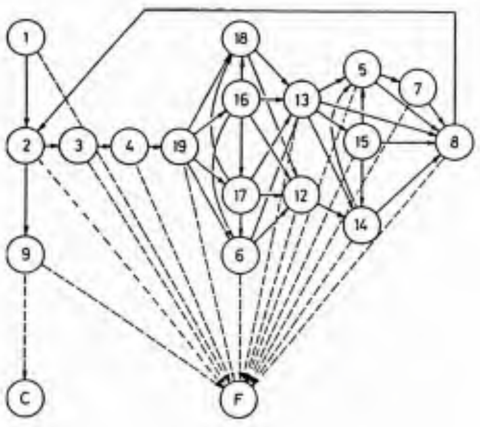


Figure XIII

The expression of the reliability is obtained from the transition matrix P according to a derivation documented in (CHE, 80).

$R = P^n (N_1, C)$ expresses the reliability of the software studied as a function of R_i and P_{ij}.

Application to the FCC, using a formal processing system (MAC, 77) makes it possible

- to determine which functions have the most adverse effect on the reliability of the whole by means of a sensitivity analysis ;
- to allocate the test effort according to the criticality of the functions ;
- to estimate the operational reliability.

An illustration for the LONDON - AMSTERDAM route is given in table VI which shows the sensitivity coefficients of the various functions together with their resulting order of importance and the distribution of three test efforts.

MODULE	SENSIBILITY	RANK	DISTRIBUTION OF TEST EFFORT $N = 10^4$	$N = 10^5$	$N = 10^6$
1	$8.6\ 10^{-4}$	2	2243	22233	75509
2-1	$8.9\ 10^{-4}$	3	2320	23011	115167
2-2	$1.1\ 10^{-3}$	1	2710	26850	134364
2-3	$1.1\ 10^{-3}$	1	2710	26850	134364
3	$8.5\ 10^{-9}$	7		64	83980
4	$4.4\ 10^{-7}$	4	8	465	148680
5	$1.4\ 10^{-10}$	11		1	7267
6	$7.7\ 10^{-10}$	13		7	18318
7	$1.3\ 10^{-10}$	10		1	7230
12	$3.2\ 10^{-9}$	6		27	55833
13	$1.1\ 10^{-9}$	5		9	24667
14	$1.1\ 10^{-10}$	8		1	7337
15	$7.6\ 10^{-10}$	12		6	18697
16	0	15			
17	$9.1\ 10^{-10}$	14		8	15791
18	$1.3\ 10^{-10}$	9		1	3616
19	$4.4\ 10^{-7}$	4	8	465	148680

Table VI

An increase in reliability results from each test phase (and the associated debugging).

After 10^4 tests, the reliability expression is :

$$R\,(10^4, n) = 0.9992365 \,/\, (1 - 5.4129624.10^{-4} \,.\, \frac{n-1}{n})\; n$$

The expressions after 10^5 and 10^6 tests are respectively :

$$R\,(10^5, n) = 0.9999229 \,/\, (1 - 0.040195288 \,.\, \frac{n-1}{n})\; n$$

$$R\,(10^5, n) = 0.99998286 \,/\, (1 - 0.99824912 \,.\, \frac{n-1}{n})\; n$$

Where n represents the number of new tests.

The family of curves presented on Figure XIV represent the phenomenon of increasing reliability for five validation stages, making a total of 40 x 10^6 tests (close to the quantity applied to the FCC in conformance with Table III). We can notice a phenomenon of Software "hardening" in proportion to the debugging effort, up to a level which gives it satisfactory reliability.

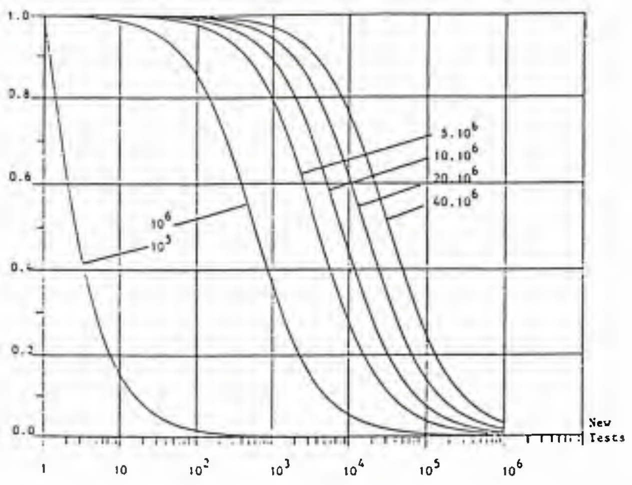


Figure XIV

CONCLUSION

This analysis clearly demonstrates the interest of software quality measurement.

Although limited to certain basic phenomena, the evaluations presented make it possible to achieve the two objectives set out in paragraph 2

- the conditions of software regression as a function of the change requests can be measured and anticipated ; in particular, this makes it possible to adapt the validation effort ;
- the adequation of production techniques to the constraints has been assessed :
    - . the modification process is the critical point in the development cycle. For this type of application with a high modification intensity, flexibility must be one of the highest-priority criteria for software design ;
    - . the test strategy (individual tests, open - or closed - loop tests) defined by the project make it possible to achieve the objective at each stage, and has proved to be effective for eliminating the various errors ;
- the advantage of a reliability model in evaluating the design and distributing the test effort has been confirmed.

Moreover, this analysis has demonstrated the interest of data collection in an industrial environment where production and quality requirements are extremely stringent. The most positive points that have been revealed are the efficiency of automated data collection in the modification phase and the sensitization of the development teams to the factors of quality.

The benefits of such an analysis are a more objective awareness of performances, a development in methods and tools, and the desire to persevere with observation and measurement.

BIBLIOGRAPHY

(ALB, 82) J.L. ALBIN, R. FERREOL
"Collecte et Analyse de Mesures de Logiciel"
TSI, Vol. 1 n° 4 - 1982 juillet-août - pp 297-313.

(BAS, 81) V.R. BASILI, D.M. WEISS
"Evaluation of a Software Requirements Document by Analysis of Change Data"
Proceeding IEEE CH 1627-9 - 1981.

(BUT, 80) S.T. BUTNER, R.K. IYER
"A Statistical Study of Reliability and System Load at SLAC"
10 th FTCS - Kyoto, Japan - September 1980.

(CHE, 80) R.C. CHEUNG
"A User Oriented Software Reliability Model"
IEEE TSE Vol. SE-6 n°2 - Mars 1980

(FEU, 79) A.R. FEUER, E.B. FOWLKESS
Relating Computer Program Maintenability to Software Measures"
National Computer Conference AFIPS 1979 pp 1003-1012.

(IYE, 82) R. IYER, S. BUTNER, E. Mc CLUSKEY
"A Statistical Failure / Load Relationship : Result of a Multicomputer Study".
IEEE-TC Vol. C31 n°7 juillet 1982.

(LIP, 79) M. LIPOW
"Prediction of Software Failures"
The Journal of Systems and Software n°1 - 1979 - pp 71-75.

(LIT, 81) B. LITTLEWOOD
"Stochastic Reliability Growth : a Model of Fault Removal in Computer Programs and Hardware designs"
IEEE - TR Vol. R-30 n°4 Octobre 1981 - pp 313-320

(MAC, 77) MACSYMA Reference Manual
The Mathlab Group - Laboratory for Computer Science MIT -version nine- December 1977 - 308 pages

(MOH, 79) S.N. MOHANTY
"Models and measurements for quality assessment of Software"
Computing Surveys, Vol. 11 n°3, September 1979, pp 251-275.

(MOT, 77) R.W. MOTLEY, W.D. BROOKS
"Statistical Prediction of Programming Errors"
RADC-TR-77-175

(MUS, 80) J. MUSA
"The Measurement and Management of Software Reliability"
Proceeding IEEE, September 1980

(NOU, 82) J. NOUBEL, R. TROY
"Evaluation de la Fiabilité du Logiciel lors du Test de Recette : Méthode et Application"
Rapport ADI-OGL - Juillet 1982

(ROM, 82) Y. ROMAIN, P. COUSTAUX
"Fiabilité des Logiciels du Centre ARGOS"
Rapport ADI-OGL - Janvier 1983

(ROS, 82) D.J. ROSSETI, R. K. IYER
"Software Related Failures on the IBM 3081 : A Relationship with System Utilization"
Compsac 82, Chicago, 8-12 november 1982, pp 45-54.

(SCH, 79) N.F. SCHNEIDEWIND
"Application of Program Graphs and Complexity Analysis to Software Development and Testing"
IEEE - T.R. vol. R-28 n°3 Août 1979

FLIGHT TEST OF A RESIDENT BACKUP SOFTWARE SYSTEM

Dwain A. Deets and Wilton P. Lock
NASA Ames Research Center
Dryden Flight Research Facility
P.O. Box 273
Edwards, California 93523-5000
U.S.A.

and

Vincent A. Megna
Charles Stark Draper Laboratory
Cambridge, Massachusets 02139
U.S.A.

SUMMARY

A new fault-tolerant system software concept employing the primary digital computers as host for the backup software portion has been implemented and flight tested in the F-8 digital fly-by-wire airplane. The system was implemented in such a way that essentially no transients occurred in transferring from primary to backup software. This was accomplished without a significant increase in the complexity of the backup software. The primary digital system was frame synchronized, which provided several advantages in implementing the resident backup software system. Since the time of the flight tests, two other flight vehicle programs have made a commitment to incorporate resident backup software similar in nature to the system described in this paper.

NOMENCLATURE

ADFRF	Ames-Dryden Flight Research Facility	IFU	interface unit
CIP	computer input panel software	I/O	input/output
CL1	control law software (includes output commands)	LED	leading edge down
CSDL	Charles Stark Draper Laboratory	LEU	leading edge up
DFBW	digital fly-by-wire	N_z	normal acceleration, g
DOWNLINK	data recording software	REBUS	resident backup software
FAIL1	location of first failure insertion point	RM1	redundancy management software (part 1)
FAIL2	location of second failure insertion point	RM2	redundancy management software (part 2)
FRR	flight readiness review	SYNCH XLINK	synchronization and cross-link software

INTRODUCTION

Digital flight control system designers have been concerned with protecting against "generic" software errors for as long as digital flight control has been in existence. The standard approach has been to provide a dissimilar hardware backup control system, usually analog in implementation. Although a dissimilar hardware backup provides a way of maintaining control of the aircraft in the event of generic software errors, it is a costly solution that fails to take advantage of the unfailed primary system hardware assets. It also requires a redundant switching mechanism to accept system commands from the active system and pass these commands to the redundant actuators. In the case of analog backups, the disadvantages can be even greater because of the need for sensor inputs that are compatible with both digital and analog systems. This usually means a separate set of sensors are needed for each system.

Several approaches have been advocated for handling software errors without requiring independent backup hardware. One approach is to utilize dissimilar software in each of the redundant hardware channels. This approach has been used with success on the A310 airbus (Ref. 1). Although successful for that application, this approach results in a system with wider tolerances on the tracking between channels that may have disadvantages in other applications. A second approach is to implement identical dissimilar software sets within each of the redundant hardware channels. This has been referred to by various names, the most common being "recovery blocks." The resident backup software (REBUS) described in this paper is of this second type. It has the advantage of minimizing hardware and retains the advantages of relatively tight tracking between the unfailed hardware channels.

Considerable debate has gone on concerning the amount of dissimilarity necessary to assure coverage for the generic software error. In this debate, a distinction is made between several classes of software errors and the amount of dissimilarity necessary for protection for each of these error classes.

One class of generic error is the design error, in which the specified design is not adequate to handle the environment encountered in the actual application. As an example, the coupling between a structural mode and the aerodynamics is sufficiently different from that simulated in ground tests that the resulting closed loop through the control system becomes unstable.

The second class of generic software error is the implementation error, which results in a system that does not meet the specified design. A particular difficulty concerns the latent system or software defect that goes undetected in testing but is exposed under certain environmental conditions experienced in flight. These implementation errors can occur at a number of different levels: in the algorithm chosen by the designer; in the way the software coder writes the source code; or in the compiler, which converts the source code to machine language. In every case, these errors would go undetected throughout ground testing and manifest themselves in an unsafe manner in actual flight. Implementation errors in the compiler are usually considered to be least likely; a compiler receives a considerable amount of implicit testing as a result of all the code that is generated and exposed to a wide range of conditions during the extensive verification and validation that are characteristic of man-rated digital fly-by-wire systems.

Even when care is taken to assure dissimilarity for each level of implementation error, precautions must be taken to assure a successful transfer to the dissimilar software because of the nature of the digital computer and the unforeseeable consequences of a software error. This is most important during initialization of the dissimilar software because vehicle state data are needed to establish proper startup conditions. The danger exists that the software error in the primary system has caused contamination of the vehicle state data, thereby raising the possibility of initializing the backup software at an unsafe condition.

Potential problems of this type have been of concern and have been some of the principal reasons that designers have chosen to use the better understood, albeit more expensive, analog backup. The flight experiment described in this paper was done to reduce the risk associated with backup software, with the hope that this more cost-effective method will be suitable for use in future flight control system designs. The experiment utilized the F-8 digital fly-by-wire (DFBW) airplane operated by the NASA Ames Research Center's Dryden Flight Research Facility (ADFRF) at Edwards, California. The system implementation was accomplished by the Charles Stark Draper Laboratory (CSDL).

FLIGHT EXPERIMENT OBJECTIVES

The F-8 DFBW airplane is operated as a flight facility for evaluating new concepts, such as REBUS, in the real flight environment. One key step in preparing for a flight test is the independent review by people with broad operational experience in flight test. At ADFRF, this review process is called the flight readiness review (FRR). The value of evaluating a system or concept in flight is that the system will, by necessity, be subjected to close scrutiny and thorough testing to receive a favorable response from the FRR committee. It will also be subjected to the actual flight environment, which will be different to some degree from that experienced in ground test. Those differences, and their effect on the system behavior, are important in understanding the concept under evaluation.

These considerations contributed to the establishment of the experiment objective: to thoroughly evaluate a high-potential approach to fault-tolerant software utilizing a dissimilar software flight control system concept, subjecting it to all the processes necessary to qualify it for flight and validating those processes through subsequent flight test.

The REBUS concept was selected for evaluation because of its system-wide efficiency gained by utilizing the same computing hardware used by the primary system.

F-8 DIGITAL FLY-BY-WIRE SYSTEM

The F-8 DFBW airplane (Fig. 1) was modified by installing a fail-operate, fail-safe digital fly-by-wire system. (A detailed description of the system is presented in Ref. 2.) It is useful to provide some details in a simplified form to make explanations of the REBUS modifications easier. The DFBW system contains a triplex digital primary system with a triplex analog computer bypass system serving as its backup. Figure 2 illustrates the interface between the two systems. The triplex primary system utilizes redundant digital computers, and CSDL provided interface units (IFUs). Redundant aircraft motion sensors and pilot stick transducers are conneced to the IFUs. Surface commands are passed through an analog midvalue voter, with the midvalue of the three channels passed on to the servo drive electronics. This midvalue voter, being external to the digital computer, has additional analog circuitry that can declare any one of the digital output commands as failed and disable the particular channel output at the midvalue input plane.

Analog Backup System

The analog backup is a direct electrical link between the pilot's stick and the servo drive electronics. The redundant commands are passed through the same midvalue select devices utilized by the primary system. This system, forward of the midvalue voter, is called the computer bypass system.

Primary System

The primary system is frame synchronized. This synchronism facilitates the exchange of data between computers and the input/output (I/O) function that distributes sensor information between computers, such that each receives a copy of all available sensor inputs.

The primary software mechanizes a number of different control laws for the aircraft. They are the direct mode (control stick to surface actuator without any feedback) and stability augmentation system mode for the pitch, roll, and yaw axes. The pitch axis also contains the command augmentation system for further improvement in handling qualities. Finally, there are the standard autopilot modes of aircraft control.

The input sensor set includes not only pilot stick and pedal sensors and vehicle motion sensors, but also information on surface commands and positions. In particular, the left and right elevator positions and the last surface command to all five control surfaces are available in the input sensor buffer located external to the digital computers in the IFU. This control surface information is useful in the initialization of the REBUS software.

The primary software contains a capability of recovering from transient errors. This capability (known as "restart") initializes the control laws after a potentially resettable failure has been indicated, and it attempts to recompute, anticipating that the failure will not reoccur. The software has been set up to recycle through the restart 10 times. If the failure still persists after 10 cycles, the channel declares a channel failure and is voted out by the remaining good channels. If one channel has already failed, the flight control system would be forced into the computer bypass system.

Of the various conditions that can create a failure condition, two cause immediate failure declarations, and one (designated "execution error") is potentially resettable. These conditions are the following:

1. Failure to perform an I/O function within 53 msec of the last I/O (the control law frame time is 20 msec)

2. Failure to update the watchdog timer

3. Execution error for 10 consecutive program iterations or once every 6 iterations for 306 iterations

The reason for attempting 10 restarts before declaring a channel failure is the time duration that can be tolerated without a surface command update: For the F-8 aircraft with stable bare airframe, a 300-msec time period can be tolerated; 10 restarts require between 100 and 300 msec, depending on where the failure indication occurs within the compute cycle.

REBUS SYSTEM DESIGN

Several tradeoffs must be made before specifying a system design. Many are similar to those made traditionally for an independent hardware backup. Issues such as complexity of the backup, criteria for automatic transfer, and interchannel synchronization must be addressed. Other issues that are specific to this experiment, in that an existing system was modified, will not be discussed.

Complexity

Selection of control laws is usually based on the minimum necessary to provide return to the base and safe landing capability. In the case of the F-8 DFBW aircraft, an unaugmented backup would be adequate to provide the necessary functionality. For this experiment, the decision was made to include slightly more capability than the bare minimum to better represent modern airplanes, which generally require augmentation to some degree. Three-axis fixed-gain rate damping was selected. Figure 3 illustrates, in the form of a block diagram, the pitch-axis stability augmentation system. As can be seen, there are several nonlinear functions, such as stick shaping and dead band. The only deviation from fixed gain is the selection of a landing approach gain set when the wing is put in the up position and the control surface authorities are modified. (The standard F-8 aircraft provides for two wing-incidence positions, the wing-up position providing improved pilot visability during the landing approach.) The other axes are similar in complexity.

The control law computation cycle could be mechanized quite simply by using a straight in-line code and very few branches. Many of the self-check functions, such as sensor redundancy management, could also be eliminated to reduce complexity. The overall complexity was reduced considerably relative to the primary system. For example, the REBUS software required less than one-tenth the memory that the primary software required. The compute cycle was selected to be 50 samples per second, the same as that of the primary system (for convenience, in that it reduced the amount of separate stability analysis and validation necessary).

Transfer Criterion

A criterion was needed for automatic transfer to backup, assuming that a generic software error could cause a situation requiring action sooner than the pilot can react. The appropriate solution was to use the channel failure declaration, which is based on repeated self-execution error indication. Failures to execute while in the control law code are representative of the failure mode for which the backup software was intended, in that the system behaves in the way that a generic software failure would be expected to manifest itself.

It is desirable to minimize transients that occur upon transfer to backup if this does not contribute to increased complexity or other undesirable effects. Given that the system failure has occurred because of a generic software error, one must also assume that there is a finite probability that the record of the aircraft state as maintained in the memory of the system computers has been degraded by the failure. Therefore, one must look elsewhere for the information required to initialize the backup software.

The only information that the backup software needs to take control of the aircraft without introducing an objectionable transient in aircraft state is the current position of the aircraft control surfaces. This allows for some small transients that result from reinitialization of any active filters and changes in loop gains from state-dependent to fixed values.

Three assumptions were made relative to initialization of the REBUS software:

1. Primary software failure occurs simultaneously in at least two channels of the system.

2. The trigger mechanism generates simultaneous pulses for at least two channels of the system.

3. At least two channels simultaneously execute the input sensor I/O function.

Given these assumptions, the backup software can be initialized to the aircraft's existing control surface commands, and thus it generates no transient other than that due to gain changes. This is possible because the simultaneous performance of the initial input sensor I/O function by the computers enables them to receive a full complement of sensor signals, that is, not only their own dedicated set but those dedicated to other computers.

Return to Primary

A major issue for any backup system designer is whether return to primary should be allowed. If the primary system has suffered a major fault, then it must be assumed that transferring back to that defective system is unsafe. On the other hand, if the backup software is not providing a controllable airplane, the pilot may want to try the primary software in a last-ditch effort. Thus, both sides of the issue could be argued without reaching a clear resolution. For an experimental system such as the F-8 REBUS, it is desirable to transfer into backup even though no serious problem exists with the primary. For test purposes, it is highly desirable to be able to transfer back and forth at will during a given flight. For these reasons, provisions were included in REBUS and the F-8 DFBW primary system for transfer back to primary from REBUS.

Interchannel Synchronization

Since the primary synchronization algorithm is a software function, the REBUS must provide a dissimilar synchronization algorithm if a synchronized backup system is to be established. The dangers associated withnot being able to synchronize upon transfer to REBUS must be considered. Also, the additional complexity associated with synchronization makes it less desirable from a system complexity standpoint. For these reasons, it was decided to establish REBUS as an asynchronous system. This decision has several ramifications (such as, no exchange of data between computers can be performed because of the complexity of adding an asynchronous data transfer capability).

Sensor cross-trapping is used in the primary system to ensure that each channel operates on identical data. With the asynchronous approach in the REBUS, each computer must operate on dedicated sensors that are independent of the operation of the other computers. Because of this interchannel independence, it is of interest in both ground and flight tests to see how much variation develops between each channel.

IMPLEMENTATION

The specific details of the REBUS implementation were strongly dependent on the peculiarities of the F-8 DFBW system. Some functions were included in the total modified DFBW system because it was a flight experiment and it was required that tests be performed easily. These details include the triggering mechanism, the approach to simulating software errors, and the status of the computer bypass system.

REBUS Triggering

The failure detection logic in the primary system causes the generation of a discrete output from the computer channel that is indicating a failure. This discrete output is processed by external circuitry unique to each channel computer. This circuitry sets a channel failure indicator, which is then passed to the other channels. This discrete was convenient for use as the trigger generator; therefore, a circuit that voted the discrete output from each computer was constructed for each channel. If two of three computers would generate a discrete, the trigger pulse would be produced for that channel. Figure 4 illustrates the relative location of the added hardware including a switching card in the IFU and a transfer circuit external to the computer. Also shown is the REBUS memory in the primary digital computer.

This trigger pulse must be introduced into the computer to initiate the execution of the backup software. The pulse could not enter as some external interrupt, for the computer could fail in such a way that all interrupts would be masked. The only "interrupt" that would be recognized by the computer at any time and under any conditions was the system reset.

Each channel has its own trigger circuit. This mechanization provides protection against inadvertent switching of the system from primary software to backup software. Two of three computers must be generating a channel fail to produce the trigger pulse. If a particular trigger circuit fails and generates a pulse when none is commanded, only that channel's computer switches to backup, and any discrepancy between its outputs and the other computers' outputs will be voted out by the external mid-value voters. Failure to generate a pulse when one is commanded will not prevent the other two computers from switching to backup software.

Software Programming

Ideally, it would have been desirable to have people other than those involved in primary software development mechanize the backup software. Also, it would have been advisable to use software development tools different than those used for the primary software. Neither one of these steps for enhanced dissimilarity were used for this experiment because of the added time and cost associated with them. There would have been no way to evaluate the effects of these additional steps within the limited scope of this experiment, so it was not deemed necessary to include this additional expense.

Simulation of Errors

For the purposes of this experiment, a simulated generic software error was needed. Several types of generic software failures were studied as to how they would manifest themselves as apparent hardware failures. It was apparent that a generic software failure that would be of sufficient severity to bring the primary system down would be such that a restart would result. This can be simulated by adding an instruction to write to a protected portion of memory. Segments of memory can be protected from being written to. Thus, the simulation of a generic software error was implemented by including a pilot-selectable branch in software that included a write instruction into a protected portion of memory. Several different points within the control law code were selected for error insertion. The error insertion function was subject to an arming device, also pilot selectable.

Computer Bypass

Since the unmodified F-8 DFBW already had a backup provided by the computer bypass system, the question arose as to whether the computer bypass function should be removed when the REBUS was added. Retaining the computer bypass would be in effect a backup on a backup. However, a strong argument existed for not removing it because of the reliability record of the specific prototype primary computers available to the F-8 program. With the computer bypass system, the total system hardware reliability is adequate for safe operation. Thus, the computer bypass system was retained for its contribution to hardware reliability.

GROUND TESTS

An important step in evaluating the REBUS system was the F-8 Ironbird simulator, which utilizes a decommissioned F-8 aircraft with a complete DFBW system installed. The aerodynamics are simulated in a general-purpose simulation computer. The specialized hardware necessary for the REBUS was implemented in brassboard versions of the interface units.

Part of the ground tests was the determination of transfer transients for a number of failure insertion points within the control law code. The only significant difference was whether the failure was inserted before or after the actuator command output. Figure 5 illustrates the situation for a typical 20-msec control compute cycle. The end of CL1 (control law software module) represents the point at which the actuator output command occurs. The first failure insertion point is represented by FAIL1 (at the end of RM1, the first part of the sensor redundancy management software module). Other software modules include SYNCH XLINK, which performs the synchronization cross-link, RM2, which is the second part of redundancy management, and DOWNLINK and CIP, which process the data recording and the pilot's computer input panel, respectively. Another failure insertion point was FAIL2, representative of a failure occurring after the surface output commands. As in the ground tests, if the failure occurred prior to the surface command update, such as at FAIL1, the aircraft sustained a longer period of time without surface commands, and the backup software was initialized with relatively old aircraft state information. Depending on the aircraft motion at the time of the failure, the transient on transfer varied. In no case was the transient considered severe.

The time period with no control, between 240 and 300 msec, is a function of the amount of time the primary system is allowed to attempt to correct itself. Figure 6 shows (in a simplified form) how a fault occurring at the end of DOWNLINK and CIP would be repeated through 10 attempts to restart while still in the primary system. For this case, the transfer time was 294 msec. For the F-8 aircraft, this time period is short enough that it does not cause any control problem. For other airplanes, the allowable time period may be much shorter, in which case a smaller value for the failure counter should be selected, assuring controllability; however, this will diminish the system's tolerance of transient failures that can be withstood without transfer to the backup.

FLIGHT TESTS

At the ADFRF flight readiness review (FRR), most of the details of the design and the results from the ground tests were reviewed without significant comment. There was one major concern raised that may have implications for others. This concerned the plans for enabling the REBUS after safely at altitude. The FRR committee questioned the advisability of taking off the first time with the REBUS enabled. The committee felt that the system should not be enabled until a safe altitude was achieved. The committee also raised a question as to the expected transient if the system was forced to REBUS at liftoff, given that the REBUS was enabled. It was finally determined that the more conservative approach was to wait until a safe altitude was achieved before enabling the system.

Once the FRR committee approved the plans, the flight experiment began. The first flight was on 23 July 1984. A summary of the flights is presented in Table 1. The emphasis of the experiment was on comparing flight with simulation. This included the tracking between channels, transfer transients, susceptibility to unwanted transfers, and general operational factors.

The tracking between channels was very close. At no time during the flight tests did drifts occur between the three channels. This was in agreement with the simulation runs conducted prior to flight.

The transfer transients were negligible, even when occurring during elevated-g maneuvers. In most cases, the transfer to REBUS could not be detected in the control surface traces. Figure 7 shows a typical time response in the pitch axis, which illustrates the excellent performance for a transfer during a 3.5-g turn.

There were no unwanted transfers encountered during the flight tests. This was to be expected because the transfer mechanism was the same as had been used to cause a transfer to computer bypass in all the previous flight tests with the F-8 DFBW airplane. There had never been a transfer to computer bypass over an eight-year period of flight testing.

From an operational standpoint, no significant concerns arose. Evaluation of the handling qualities for the REBUS control laws was included under operational assessment. Two pilots evaluated the system, evaluating it as acceptable for emergency operations and preferable to the computer bypass mode.

EXTENSION OF CONCEPT TO OTHER APPLICATIONS

The experience gained with the F-8 REBUS flight experiment has raised some issues relative to future applications. Many of the recommendations that could be made apply to dissimilar backup systems in general. We will restrict the comments here to issues applicable only to resident software backups.

Much of the REBUS design was dependent on the synchronous nature of the primary system. If the primary system were asynchronous, it would be valuable for initializing the REBUS to have all surface positions available in the input data set.

Concepts similar to REBUS have already been incorporated into the designs for other flight applications. The X-wing rotorcraft program (Ref. 3) utilizes a jam-transfer software backup that is nearly the same as REBUS. The F-16C and D aircraft with digital fly-by-wire systems will have resident software backups in the primary system memories.

CONCLUSIONS

The F-8 resident backup software (REBUS) flight experiment has demonstrated a cost-effective approach to providing dissimilar redundancy to protect against generic software failures. The major findings are as follows:

1. Resident backup software that provides protection for primary software errors can be implemented.

2. Transients that occur during transfer from primary to backup can be minimized with little impact on system complexity.

3. Independent reviewers with broad operational background in flight test can be satisfied that resident software backups offer adequate assurance of flight safety.

Recent incorporation of backup software approaches similar to REBUS into the designs of upcoming flight vehicles, coupled with the successful results from this flight experiment, offers evidence that this concept will find industry-wide acceptance as a viable solution to the generic software error problem.

REFERENCES

1. Hills, A.D.: Digital Fly-By-Wire Experience. Fault Tolerant Hardware/Software Architecture for Flight Critical Function, AGARD-LS-143, Paper 2, Sept. 1985.

2. Szalai, Kenneth J.; Jarvis, Calvin R.; Krier, Gary E.; Megna, Vincent A.; Brock, Larry D.; and O'Donnell, Robert N.: Digital Fly-By-Wire Flight Control Validation Experience. NASA TM-72860, 1978.

3. Guertin, R.G.: Development Status of the RSRA/X-Wing — Rotor System Research Aircraft. AIAA-85-4008, AIAA Aircraft Design Systems and Operations Meeting, Colorado Springs, Colorado, Oct. 14-16, 1985.

Table 1. REBUS FLIGHT SUMMARY

Number of flights in REBUS	6
Total flight time for these 6 flights	6 h 50 min
Total flight time in REBUS	3 h 54 min
Number of transfers to REBUS	22
Number of transfers to primary	18
Number of transfers at >1 g	6
Highest g level at REBUS transfer	3.5 g
Number of low approaches in REBUS	6
Number of touch and gos in REBUS	10
Number of landings in REBUS	5

ECN 3312

Figure 1. F-8 digital fly-by-wire airplane.

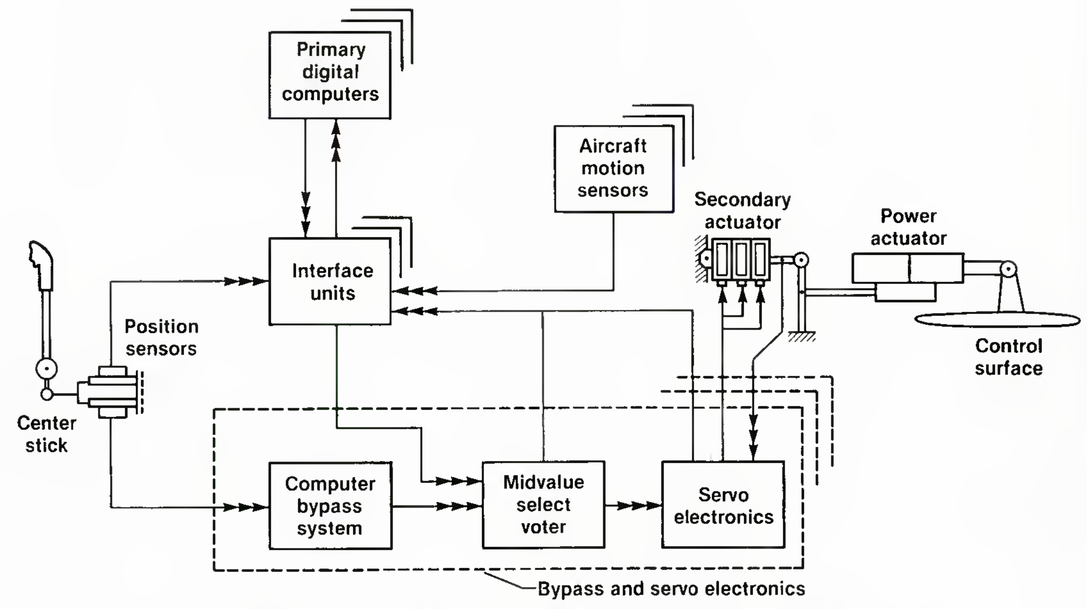


Figure 2. F-8 DFBW simplified control system schematic.

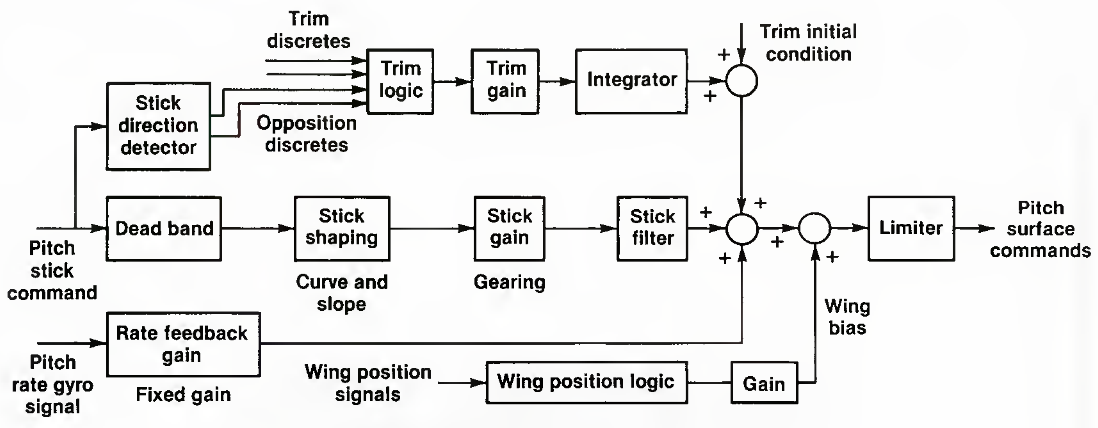


Figure 3. REBUS pitch-axis control law block diagram.

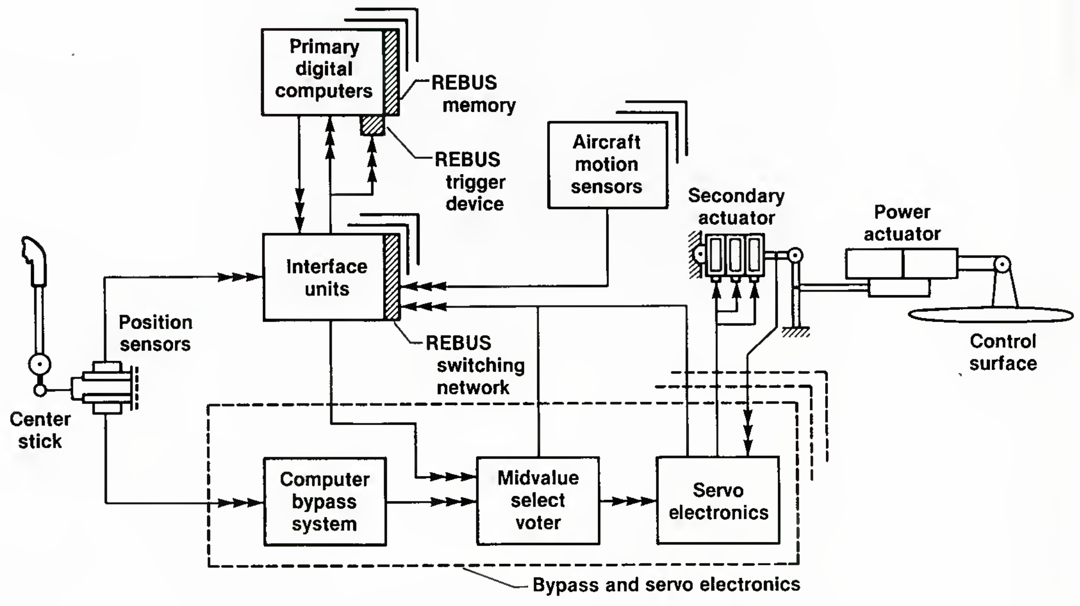


Figure 4. Modified system for REBUS implementation.

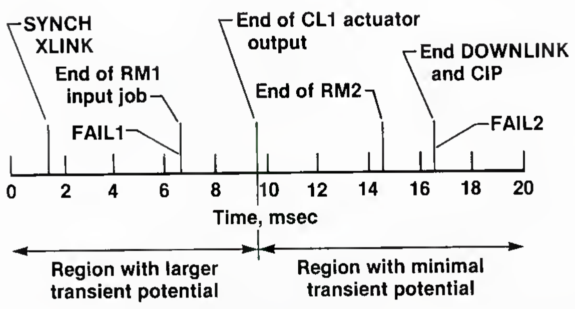


Figure 5. Failure insertion placement and its effect on transfer transient.

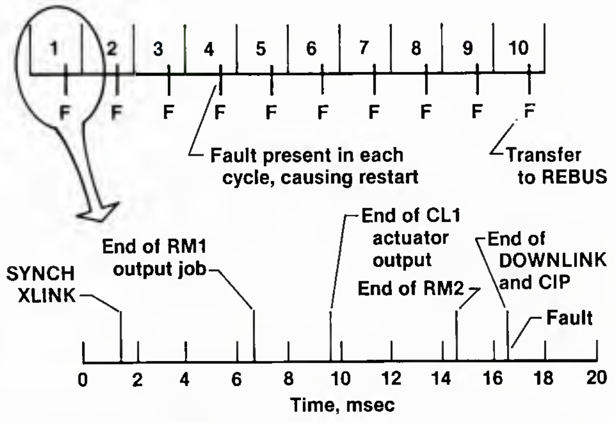


Figure 6. Timing diagram for fault insertion late in minor cycle.

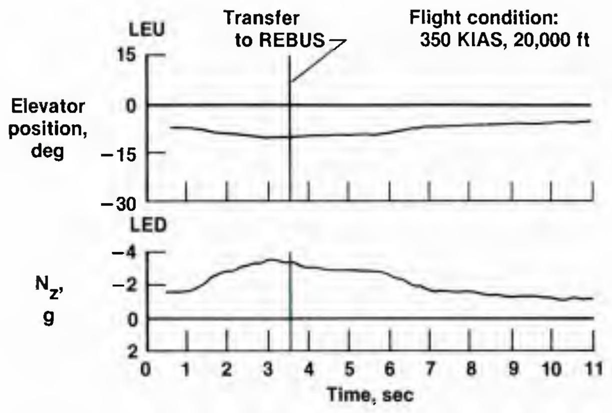


Figure 7. Flight-measured time response for elevated-g transfer to REBUS.

AGARDograph No.289
Advisory Group for Aerospace Research and Development, NATO
FAULT TOLERANT CONSIDERATIONS AND METHODS FOR GUIDANCE AND CONTROL SYSTEMS
Edited by M. l'Ingénieur Général Marc J.Pelegrin
Published July 1987
136 pages

This AGARDograph, prepared and edited at the request of the Guidance and Control Panel of AGARD, presents a description of recent trends and developments in analysis and design methods for fault tolerant guidance and control systems architectures. The problems and issues associated with these architectures, including both hardware and

P.T.O

AGARD-AG-289

High integrity software
Fault tolerance, fault avoidance software
Software dependability
Fault diagnostic
Error recovery
Dissimilar redundancy

AGARDograph No.289
Advisory Group for Aerospace Research and Development, NATO
FAULT TOLERANT CONSIDERATIONS AND METHODS FOR GUIDANCE AND CONTROL SYSTEMS
Edited by M. l'Ingénieur Général Marc J.Pelegrin
Published July 1987
136 pages

This AGARDograph, prepared and edited at the request of the Guidance and Control Panel of AGARD, presents a description of recent trends and developments in analysis and design methods for fault tolerant guidance and control systems architectures. The problems and issues associated with these architectures, including both hardware and

P.T.O

AGARD-AG-289

High integrity software
Fault tolerance, fault avoidance software
Software dependability
Fault diagnostic
Error recovery
Dissimilar redundancy

AGARDograph No.289
Advisory Group for Aerospace Research and Development, NATO
FAULT TOLERANT CONSIDERATIONS AND METHODS FOR GUIDANCE AND CONTROL SYSTEMS
Edited by M. l'Ingénieur Général Marc J.Pelegrin
Published July 1987
136 pages

This AGARDograph, prepared and edited at the request of the Guidance and Control Panel of AGARD, presents a description of recent trends and developments in analysis and design methods for fault tolerant guidance and control systems architectures. The problems and issues associated with these architectures, including both hardware and

P.T.O

AGARD-AG-289

High integrity software
Fault tolerance, fault avoidance software
Software dependability
Fault diagnostic
Error recovery
Dissimilar redundancy

AGARDograph No.289
Advisory Group for Aerospace Research and Development, NATO
FAULT TOLERANT CONSIDERATIONS AND METHODS FOR GUIDANCE AND CONTROL SYSTEMS
Edited by M. l'Ingénieur Général Marc J.Pelegrin
Published July 1987
136 pages

This AGARDograph, prepared and edited at the request of the Guidance and Control Panel of AGARD, presents a description of recent trends and developments in analysis and design methods for fault tolerant guidance and control systems architectures. The problems and issues associated with these architectures, including both hardware and

P.T.O

AGARD-AG-289

High integrity software
Fault tolerance, fault avoidance software
Software dependability
Fault diagnostic
Error recovery
Dissimilar redundancy

software aspects, are addressed exploring the many developments underway in NATO in the following areas: Advances in fault tolerant architectures; Advances in analytical fault-detection methods; Design considerations and methods; Analysis and testing methods.

This AGARDograph has been prepared at the request of the Guidance and Control Panel of AGARD.

ISBN 92-835-1554-4

software aspects, are addressed exploring the many developments underway in NATO in the following areas: Advances in fault tolerant architectures; Advances in analytical fault-detection methods; Design considerations and methods; Analysis and testing methods.

This AGARDograph has been prepared at the request of the Guidance and Control Panel of AGARD.

ISBN 92-835-1554-4

software aspects, are addressed exploring the many developments underway in NATO in the following areas: Advances in fault tolerant architectures; Advances in analytical fault-detection methods; Design considerations and methods; Analysis and testing methods.

This AGARDograph has been prepared at the request of the Guidance and Control Panel of AGARD.

ISBN 92-835-1554-4

software aspects, are addressed exploring the many developments underway in NATO in the following areas: Advances in fault tolerant architectures; Advances in analytical fault-detection methods; Design considerations and methods; Analysis and testing methods.

This AGARDograph has been prepared at the request of the Guidance and Control Panel of AGARD.

ISBN 92-835-1554-4

AGARD

NATO OTAN

7 rue Ancelle · 92200 NEUILLY-SUR-SEINE

FRANCE

Telephone (1)47.38.57.00 · Telex 610 176

DISTRIBUTION OF UNCLASSIFIED AGARD PUBLICATIONS

AGARD does NOT hold stocks of AGARD publications at the above address for general distribution. Initial distribution of AGARD publications is made to AGARD Member Nations through the following National Distribution Centres.Further copies are sometimes available from these Centres, but if not may be purchased in Microfiche or Photocopy form from the Purchase Agencies listed below.

NATIONAL DISTRIBUTION CENTRES

BELGIUM
Coordonnateur AGARD — VSL
Etat-Major de la Force Aérienne
Quartier Reine Elisabeth
Rue d'Evere, 1140 Bruxelles

CANADA
Defence S
Dept of
Ottawa

DENMARK
Danish
Ved Id
2100 C

FRANCE
O.N.E.
29 Ave
92320

GERMANY
Fachin
Physik
Kernfo
D-751

GREECE
Hellen
Resear
Holarg

ICELAND
Director of Aviation
c/o Flugrad
Reyjavik

ITALY
Aeronautica Militare
Ufficio del Delegato Nazionale all'AGARD
3 Piazzale Adenauer
00144 Roma/EUR

LUXEMBOURG
See Belgium

R

ishment

AGARD

UNITED KINGDOM
Defence Research Information Centre
Kentigern House
65 Brown Street
Glasgow G2 8EX

UNITED STATES
National Aeronautics and Space Administration (NASA)
Langley Research Center
M/S 180
Hampton, Virginia 23665

THE UNITED STATES NATIONAL DISTRIBUTION CENTRE (NASA) DOES NOT HOLD STOCKS OF AGARD PUBLICATIONS, AND APPLICATIONS FOR COPIES SHOULD BE MADE DIRECT TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS) AT THE ADDRESS BELOW.

PURCHASE AGENCIES

National Technical
Information Service (NTIS)
5285 Port Royal Road
Springfield
Virginia 22161, USA

ESA/Information Retrieval Service
European Space Agency
10, rue Mario Nikis
75015 Paris, France

The British Library
Document Supply Division
Boston Spa, Wetherby
West Yorkshire LS23 7BQ
England

Requests for microfiche or photocopies of AGARD documents should include the AGARD serial number, title, author or editor, and publication date. Requests to NTIS should include the NASA accession report number. Full bibliographical references and abstracts of AGARD publications are given in the following journals:

Scientific and Technical Aerospace Reports (STAR)
published by NASA Scientific and Technical
Information Branch
NASA Headquarters (NIT-40)
Washington D.C. 20546, USA

Government Reports Announcements (GRA)
published by the National Technical
Information Services, Springfield
Virginia 22161, USA

Printed by Specialised Printing Services Limited
40 Chigwell Lane, Loughton, Essex IG10 3TZ

ISBN 92-835-1554-4